Contents

0.1 Foreword

This report has been written to conclude a semester project for the Operating Systems Laboratory (LSE) of the Swiss Federal Institute of Technology (EPFL) in Lausanne. The aim of this semester project was to explore Jini technology, to highlight the interesting features it provided, and to detail a practical example of Jini ”at work”.

This work is first aimed at presenting the innovative tool Jini is. However, I also hope that this report will encourage and help readers to practically use Jini to build new robust and reliable networks!

0.2 The beginning...

In 1998, Sun Microsystems announced publicly the existence of the Jini project. They presented Jini as a set of Java classes that could potentially change the way programmers used to build networks. In fact, the technological concepts behind these few classes could really help to build robust, reliable and especially polyvalent networked systems.

To ensure anyone could participate in the evolution of Jini, Sun immediately placed the development of this project under the Sun Community Source License (SCSL). A direct consequence of this decision is that many private programmers have worked for the last three years to improve Jini features. The result is a powerful, robust and reliable tool, that should be considered by anyone building a network.

This report is a quick overview of Jini technology, that is structurated around three main parts:

1. A Jini presentation intending to describe Jini and to explain the key concepts Sun’s engineers used to make Jini a powerful tool.
2. A practical example illustrating the model.
3. An appendix providing some useful references and advise about installing and using Jini.

0.3 Acknowledgements

I would like to thank LSE semester projects supervisors Pr. A. Schiper and M. Wiesmann. They gave me the opportunity to explore the various aspects of Jini, and I hope that my work will be useful for their laboratory. A special thank to Mr Wiesmann as he helped to structurate my work as well as this report.

Part I

Presentation of Jini

Chapter 1

Introduction

1.1 What is networking nowadays?

During the last ten years, computer networks have grown all over the world. The number of devices connected together increases every day, and new technologies appear more and more often.

Some of the main problems of network conception have been given solutions. Bandwidth has drastically increased on local networks, TCP/IP and UDP/IP eliminated many other protocols, and became standard. An important example is about Java. With Java, not only data could move to the operational code, but also the code could go to data. This property is usually called ”portability” of the code. RMI and CORBA provided a way to bring OO programming on the network.

While many problems have been resolved, either by hardware improvements or by software engineering, some crucial features of information networks still need strong development. One of the most important concern is about networks' robustness. A robust system doesn't need too frequent administration tasks, reducing costs while making it possible to rely on the existing infrastructure even when nobody is there to maintain the network. This is a real requirement as the size of the networks is growing rapidly and as the users know less and less about the technologies they use.

As it is impossible to eliminate network errors and failures, today networks should be able to keep working correctly even after an error. To achieve this, the networked entities should be aware of their environment at any time. Unfortunately, it is actually not so easy to implement such a property, and existing networks cannot be all changed into a brand new network type. The most efficient way is therefore to try to add existing networks new useful features.

The easiest method to improve the existing technologies is certainly to try first to list the most common causes of troubles, then to give as many individual solutions as

possible to each problem, and eventually to group these solutions in a new technology. That's what Sun Microsystems' engineers did, and in 1998, Jini was born...

They first noticed that, interestingly, in many existing networks, the action to connect new devices is a source of problems. Larger the network is, less reliable it is. They thought that things should not be like that, as add new devices on a network should increase its redundancy, reducing the risks of failure. So the first problems listed by Sun's engineers were:

- The complexity of existing network systems.
- The poor scalability of some network types.
- The lack of reliability.

Therefore, they decided to build simple, scalable and reliable networks.

Simplicity To make a network simple is essentially consisting in making it easy to use and to administrate. The best technique is, to reduce users' interventions to the minimum, (i.e. plug and turn on/off the networked devices) and to automate at most administration.

Scalability To make a network scalable is mainly a problem of design. The most common solution is to federate smaller networks into a bigger system, which might itself be federated into another system, and so on. Create this kind of hierarchies presents many advantages, and often allows to build networks that fit with the reality (a computer is in a room, a room is in a building, a building is in a city, ...).

Reliability To make a network reliable is much more difficult, as the set of possible troubles is unbounded. Partial failures are particularly insidious, as they might not be detected by the unaffected part of the network. When using distributed systems, how to distinguish a failed component from a slow component? It is often impossible to even determine if an error happens! Therefore, to ensure that consistency among networked participants is maintained is a very hard task, and an important issue.

1.2 What is Jini?

Jini is a technology which aims to make networks robust, scalable and easy to use. These networks should allow any type of electronic device and any piece of software to cooperate in a natural way. The basic idea is that users should just know how to turn on a device. Then, things should just work.

To achieve this, Sun's engineers agreed on three key features of a Jini-enabled network:

1. Spontaneous networking: No user intervention is required when services or devices are brought on- or offline.
2. Self-healing: The networked environment can adapt to its constantly evoluting structure.
3. Agnostism: Consumers of Jini services need no prior knowledge of services' implementations.

Before these three concepts are explained, it is necessary to quickly define a few terms, that will all be defined again in more detail in the Definitions chapter:

Network Though the Jini concepts are general enough to potentially adapt to almost any type of network, Jini uses TCP/IP. Provided that this report is practise-oriented, we will limit ourselves to TCP/IP networks.

Service In Jini, anything that can be used through the network is called service. The fact that a service is implemented by hardware or software is irrevelant, and there is no distinction between the "print" services a printer or a TeX-renderer provide.

Client The user of a service is called a client. A service might be itself client of another service. This unique definition is a first step toward simplicity.

Community (or djiin) The services are grouped in cooperating sets called communities (or djiins). Any service is a member of at least one community.

Federation Communities may be linked together into larger groups called federations. This way of addressing scalability is simple and efficient.

Now that these few words have been introduced, it is possible to come back to the three key features of a Jini network:

Spontaneous networking Spontaneous networking is the ability of each service to join the networked community by itself, and the ability of the community to detect when a service (dis)connects. That gives the community the possibility to know at any time which services are available on the network.

Self-healing Self-healing is the ability of the community to repair itself when an error occurs. That is, guarantee that the community will always recover a consistent state after any error on the network. That also involves high redundancy to reduce the dependence towards key machines.

Agnostism Agnostism is the ability for clients to use a service through its interface without any knowledge about the implementation of the service (which might be hardware or software-based). That allows clients to use any type of service transparently.

The next section describes the underlying mechanisms used by Jini to support the previous features.

1.3 How does it work?

The mechanisms used by Jini to support spontaneous networking and self-healing are mainly implemented through four concepts:

1. Discovery (Spontaneous networking)
2. Lookup (Spontaneous networking, self-healing)
3. Leasing (Self-healing)
4. Remote Events (Spontaneous networking)

Discovery Discovery defines how Jini services find the communities reachable through the network. Once a service has completed discovery, it will know where the lookup services are.

Lookup Lookup refers to the lookup service as well as what a service can do with the lookup service. Basically, the lookup service is a service that registers all the services connected to the network, and stores some information about what these services do. Usually, services first register with the lookup (except for clients that do not provide any service), and then use it to know what are the available services. As the services cannot be found without registering with the lookup, they are all clients of at least one lookup service.

The combination of discovery and lookup are the way Jini implements the basis of spontanteous networking.

Leasing Self-healing is achieved through leasing. A lease is actually an amount of time a service can grant to its clients to use it. The client must then renew the lease as long as it wants to use the service. If a client crashes, it won't be able to renew its leases, and the services will know that the client is not working (correctly) any more. This ensures that any network failure is detected at worst after the time of the longer lease. In particular, the lookup services uses leases to keep the list of the services correct, so that the combination of leases and lookup are the way Jini implements self-healing.

Remote events Remote events are used by services to notify other services when some interesting things happen over the network. Only the clients that explicitly ask a service to be notified receive the events this service throws (this is known as Publish-Subscribe paradigm). Obviously, the services that throw events use leasing to avoid sending events to crashed clients. An advanced feature of Jini spontaneous networking is the ability to for the clients of a lookup to be notified when a given service connects to the network. When the given service registers with the lookup, the lookup will send an event to the client, saying that the requested service is available. Remote events are therefore an important element of spontaneous networking too.

Agnostism The key idea is that only the service knows about its own implementation. Consequently, only the service itself may know about how it has to be used. Therefore, a simple solution is to store the service's driver in the service itself. As this driver enables the service to be controlled through the network, it is usually called a Jini proxy. But how do clients get the Jini proxy? When joining the lookup, services simply upload their Jini proxies to the lookup when they perform registrar. Then, these Jini proxies are downloaded by the clients from the lookup. What these Jini proxies actually are is detailed in chapter 2 as section 3.2describes how the clients ask the lookup for a specific type of service.

1.4 Other features

Jini foundations define a few other interesting features, and provide some useful tools. However, none of these is as fundamental concept as transactions.

Transactions Transactions are used to ensure data integrity. A service may want to be able to complete either all operations grouped under a transaction, or none of them. The most common example is the transfer of money between two bank accounts. As the accounts might be stored on different machines, a partial failure might occur, allowing one account to in- or decremented as the other one remains unchanged. It is obvious that the system must ensure that either both operations are performed, or nothing is done. The solution is to group both operations into one transaction. This is called atomicity, because the operations connot be performed separately. After a transaction has completed, the participants must be in a consistent state. That means data they contain respect some integrity constraints (a weight cannot be negative, ...). That's called consistency. As concurrent transactions should not affect each other as long as they haven't completed, it is necessary to isolate them, which is called isolation. Eventually, once a transaction successfully completed, it is obvious that all of the changes have to be made permanent, which is referred to as durability. The properties of transactions, i.e. atomicity, consistency, isolation and durability are usually called ACID properties.

Transactions are detailed in the dedicated section of the chapter Advanced Jini, as well as some other useful features Jini defines.

1.5 A typical example

A common example that illustrates Jini potential is about hardware co-operation.

Imagine that we have a local TCP/IP network grouping a few computers, and suppose a lookup service is hosted by one of these machines.

User Alice plugs her Jini-enabled digital camera on the network. Her camera will see that there is no available printing services, but it will also be informed that some machines provide a storage service for her pictures. Another computer could also host an e-mail service, allowing the camera to directly send the pictures via SMTP. As there is no active printer, Alice won't be able to print her pictures, and she will therefore ask the lookup to be notified when a printer appears on the network.

User Bob is running a navigator, and wants to print some web pages. As the "print" button of his software is unfortunately inactive, he also asks to be notified when a print service will be available.

Once a printer eventually appears on the network, it will first register with the lookup. As soon as the lookup knows that a printer has been plugged, it will notify both the camera and the navigator that a printer is available. Alice will then be able to print her pictures and Bob will be able to print his web pages.

Obviously, if the camera is unplugged before the printer appears, the lookup will notice it, and won't spend resources trying to notify the camera.

This short example illustrates how easy to use Jini networks are. Users just plug and unplug their devices, they just use the available softwares, and ask the lookups for services. No prior knowledge about the type of device that will implement a service is required, and no installation of drivers is necessary. Everything is just plug and run!

Chapter 2

Definitions and technical requirements

2.1 About definitions

Jini is a recent technology, and is evoluting quickly. Many new features appear with each new release of Jini, and new concepts are implemented in each version. However, the structure of Jini networks hasn't changed since Jini was first released.

The following definitions help to understand how Jini networks are structurated. As the meaning of some words might differ from other definitions found in the literature, these definitions will serve as a reference for this report.

The second part of this chapter describes how Jini services are described and used.

2.2 Structure

Network

As the networks Jini can use have to support TCP/IP multicast, the word network will always refer to TCP/IP networks.

2.2.1 Device

A device is any piece of electronical hardware, from the electrical switch to the supercomputer. Unless they are machines themselves, services have to be connected to at least one machine. The connection may be of any type, from serial cable to infrared wireless connection, or even smoke signals, that is totally hidden from Jini.

2.2.2 Machine

A machine is a device that has enough computational abilities to run a JVM, or at least a subset of Java (eJava, pJava, ...). A machine has to implement TCP/IP, and be able to be plugged on the network. For example, PC's and Java-enabled PDA's are machines.

2.2.3 Service

A service is anything that can be used by clients over the network. Concretely, services may be devices as well as software.

As a service must at least be able to find a lookup and to register with it by providing its Jini proxy, services must be hosted on machines (In fact, to complete lookup and discovery, the service must use Java, so that the device hosting the service must have a JVM). The service must provide a proxy that enables the clients to use it (this proxy may not be written exclusively in Java).

This is quite a heavy set of conditions, while Jini had been designed to be used with any type of device. This means that a simple electrical switch should fulfil the basic conditions to be a Jini service. It is clearly impossible to embed even a partial JVM in every switch we want to control through a Jini network. Obviously, this is not the case, as to solve this problem, Sun's engineers agreed on the principle of delegation. This principle allows a service to delegate any task it has to fulfil to another machine on its behalf. That means that our switch may be seen as a Jini service if a corresponding piece of software is running on any machine over the network. To be useful, this piece of software should control the switch, but that is not done under the responsibility of Jini. The only things this small program has to do from Jini point of view is to participate in discovery, to register with at least a lookup, and to provide a downloadable proxy allowing to control the switch, either directly, or through any machine on the network.

To summarize, the minimal requirements to meet the "service" definition of Jini are:

- The service must be able to connect to a TCP/IP network (with full multicast abilities), either by itself, or via a delegate on its behalf.
- The service or its delegate must participate in the discovery process to find a lookup. It must then register with the lookup by providing a downloadable proxy.
- Eventually, even if it isn't a strict requirement, a service should almost always be able to renew its leases (if it doesn't, it will systematically be rejected by the network after its leases expire).

2.2.4 Delegate

A delegate is a piece of software that is hosted on a machine, and that is related to a service. The tasks of a delegate are to perform operations for one or more services.

Delegates are especially useful to services that are concretely implemented by simple devices. For example, a switch or a simple printer will often use a delegate to perform discovery, lookup and lease renewal. The only thing that will link the client to the device is the Jini proxy the delegate provides to the lookup. With this proxy, clients should be able to use the device, but Jini doesn't specify how. For example, the printer Jini proxy will probably be a driver that will control directly the printer via the network. For the switch, that might not be plugged on the network, the Jini proxy might call a computer controlling the switch. The only technical requirement a device has to fulfil to use delegates is to be actually connected to the network, either directly (printer), or via another device (the speakers of a computer).

Delegates may also be used to perform any operation for other services. For example, some delegate may renew leases for one or more services, that might be running or not on the same machine. In fact, any standard Jini operation, such as registering with the lookups or listening for remote events, can be delegated. However, the provided Jini proxy eventually has to actually access the service itself.

The consequence of using delegates may vary widely. If a service uses a delegate located on a remote machine to renew its leases, it will be disconnected whenever the remote machine has a problem. Even worse, if the remote machine performs lease renewal for several services, its single crash will result in all the services to be disconnected. However, a delegate that would run on the same machine as a service would not increase the risk of failure.

Another possible consequence, is that there is no self-healing guarantee. A programmer could just decide not to implement anything to check the service is still alive, and simply tell the delegate to always renew the leases of the service. This might lead to bad errors when trying to use the Jini proxy of the service. These issues should always be considered by the programmers, and delegates should find a way to periodically check that the services they represent are still alive.

2.2.5 Client

A client is a piece of software that uses a Jini service through the network. By extension, a person that uses a service is also called a client, but this person actually interacts with the service via the client software. All the services must be clients of the lookup service. As all the lookup services themselves should register with each other, so that it is correct to notice that every service is a client.

To make it possible for a piece of software to use a Jini service, this piece of software is required to implement a few requirements:

- The client must be able to find a lookup service. So it must use the discovery protocol.
- The client has to download the proxy, and must then be able to run it. As proxies have to use at least Java, the device on which the client is running must have a JVM.
- In many cases, the client wishes to be notified about changes over network, and has therefore to use remote events.

Clients may use delegation as services do.

2.2.6 Community

As mentioned in section 1.2, a community is a cooperating set of Jini services.

We will see in the chapter about discovery that lookup shall be found through two protocols: the unicast and the multicast discovery protocols. The first protocol allows a client to find a specific lookup anywhere over the network using its IP address (or its URL). That involves that the lookup location is known, and this is not the common case. Therefore the second protocol is much widerly used, as it allows to find lookups without knowing their addresses. As its name indicates, this protocol uses multicast calls to find a maximum number of lookup services over the network.

Consequently, the interesting part of the network is the multicast subspace around the client. And the community usually covers this network area, grouping together all the services plugged over the multicast subset of the network.

The minimal requirements to create a community are very simple: one or more lookup services must run on a multicast-enabled TCP/IP network.

2.2.7 Federation

A federation is used to group communities. Rather than extending the community boundaries, Jini offers this concept to allow communities to communicate with each other. As the federated communities are usually not on the same multicast space, they must use unicast calls to communicate. The trivial consequence is that the network is not spontaneous anymore, as the lookup services of the different communities have to know the addresses of the other lookups. Nevertheless, as lookup services register with each other using leases, failure detection is still ensured (if the connection is lost, the lease won't be renewed).

The concept of federation is widely used and allows communities to be spontaneous and self-healing, while allowing clients to use "remote" services that run on

other communities. It is usually implemented by registering only the lookup services of each community with each other via the unicast discovery protocol. Any client may then ask the lookups of its own community the addresses of the remote lookups, and access these lookups via the unicast discovery protocol.

The topologies of the federations are not defined by Jini, and it is up to the administrators to design the most appropriate structure. The three main options are Hierarchies, Stars and Islands. To summarize, even if the fact of federating communities requires some administration and is not spontaneous, it is worth using it widely to build large networks.

2.2.8 Groups

A community usually covers the entire multicast subspace of a network. Sometimes, it might be useful to segregate the services of the community, especially if there are numerous services, and that is made possible by groups. Although some books define a group as a community, it would be more exact to define a group as a subset of the services of the community. It consists in a name, that is used to group a subset of services together. These service may or may not be available for the rest of the community.

In practice, a community knows at least two groups, the ”public” group, and the ”none” group. The first is the default group. The second is the empty subset of services of the community. The various lookup services over the network have to listen for one or more specific groups. If someone, for example, wants to network Jini services for experimental purposes, this person will certainly create a group called ”test”, this group should not interact with the other groups. To make it possible, it will be necessary to run one or more lookup services that listen to the ”test” group. When participating in the discovery process, a service (or a client) may ask to find lookups listening for a specific group.

By default, the lookup services listen to the ”public” group. If another group is created, some lookup must listen for this group. A lookup might support several groups, but Sun’s implementation of Jini is not able to segregate the groups if a lookup listens to several of them. To really separate the services, it is therefore necessary to run independent lookup services for each group (once this is the case, a group has the same properties a community would have).

2.3 Jini proxies

A Jini proxy is a downloadable proxy that should be able to communicate with the corresponding service. The way this communication is implemented is not defined by Jini. What is defined by Jini is only that a Jini proxy is a serialized Java object. This

object may be an RMI stub, it might be a Java program that uses a private communication protocol, it might even be a Java program that contains and launches an embedded program written in another language.

An important consideration is about the way our service should actually use the Jini proxy it has dowloaded. The most common case is that our client knows an interface the proxy implements, and is therefore able to use the proxy via this interface. Another possible case is that our proxy contains a graphical interface, allowing a human user to control the service through the network. Java beans also provide a way to make it easier for a client to use the Jini proxy. If none of these conditions are met, our client will probably not be able to use the service.

Sun is working with many companies to define a set of common interfaces to use all the common network devices and services. Standardization is a long process, but we will hopefully be able to use standard interfaces very soon to access printers, cellular phones, routers, etc.

How Jini proxies are actually found by the clients in detail in section 3.2.2.

Chapter 3

Fundamental Concepts

3.1 Discovery

3.1.1 Concept

Discovery is a process that allows services to find lookup services over the network. This is done via the discovery protocol. Once one or more lookup have been found, the service can perform the join protocol to become available for the clients over the network.

Discovery is the first stage a service has to complete to interact with the Jini world, and should therefore be part of the boot sequence of Jini-enabled devices/services.

3.1.2 Implementation overview

There are two main protocols: the multicast discovery protocol and the unicast discovery protocol.

The first one is intended to find the active lookups of the community. This protocol is implemented by Jini core classes.

The second one is useful to register with ”remote” lookups. That is, lookup services that are running outside the multicast subspace of the network. To access a ”remote” lookup, a service must know its URL using the jini syntax (e.g. jini://insun3.epfl.ch:2179).

The result of the discovery protocols is a list of references to lookup services.

A special case not covered by these protocols is when a lookup service appears in the community. It should be possible for this lookup to announce its arrival not only to the other lookups, but also to all the other services. This is made possible by the multicast announcement protocol, the third and last of the discovery protocols. To

distinguish clearly the multicast protocols, some books refer to the multicast discovery protocol as the multicast request protocol. In this report, I the following conventions will be used:

- MultiCast Request Protocol or multicast discovery protocol will be referred to as MCRP
- MultiCast Announcement Protocol as MCAP
- UniCast Discovery Protocol as UCDP

The aim of this work is to give an overview of Jini features, and it is therefore beyond the scope of this report to treat all the specific techniques that may be used by discovery. However, there are two quite common cases that are important enough to be mentioned here:

- Segregation of services using groups: When performing MCRP while using groups other than the default ”public”, it is necessary to specify explicitly the group names to find the corresponding lookup services.
- Segregation of users using rights: It is possible to create lookups with different authorisations within a single group. To access these specific lookups, it is usually easier to use the UCDP.

3.2 Lookup

3.2.1 Concept

The lookup service is a service that references all available services. It offers clients two ways to formulate requests:

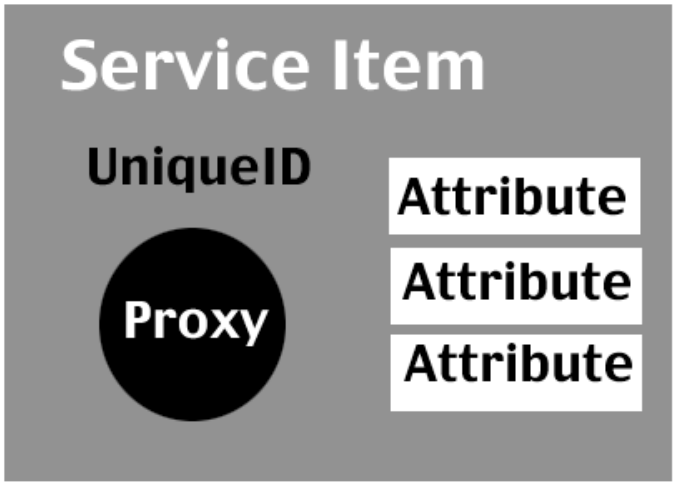
- Service-definition based requests: requests based on service ID or classname (which is known as white pages service).
- Service-function based requests: requests based on some service’s attributes (which is known as yellow pages service).

The lookup is also intended to provide services and clients all the necessary infrastructure to meet each other and to initiate their interaction. That is, explicitly, to provide the services a space and well-defined way to store their proxies, and to provide the clients the proxies they are looking for. If the requested service is not found, the lookup should provide the client a way to ask to be notified when the service appears.

The lookup service is the only service that must run in the Jini community. Consequently, the lookup service is an especially crucial failure point in the network. That’s why high redundancy and geographical dispatch of lookup services is recommended. That’s also why service should always register with a maximum number of lookups, and should be able to register with any new lookup that uses MCAP. That’s the key feature to enable self-healing in a system.

3.2.2 ServiceItem:

The lookup basically keeps a list of things associated with each of the services it references. These things are the Jini proxy, a unique service ID, and some attributes to describe the service. They are grouped in an object implementing the net.jini.core.lookup.ServiceItem interface.



Attribute: An attribute is a Java object attached to a service proxy. The main purposes of attributes are:

- To provide useful information about the service.
- To allow the lookup to compare services attributes with client-requested attributes, to determine whether a service matches an attribute-based request.

Attributes are serialized and stored on the lookup services with their corresponding Jini proxy. An attribute object must implement the net.jini.core.entry.Entry interface, that extends the Serializable interface.

Attributes may be controlled (or changed) by the service exclusively (serial number, ...), or by the service as well as by the users of the service (location of the service,...). An attribute that is exclusively controlled by the service will implement the net.jini.lookup.entry.ServiceControlled interface, and is said to be service controlled (SC).

Jini defines some standard attributes, such as:

Address Street, city and country, not SC.

Comment not SC.

Location location the service within an organisation (building, room, ...), not SC.

Name Understandable name for users, not SC.

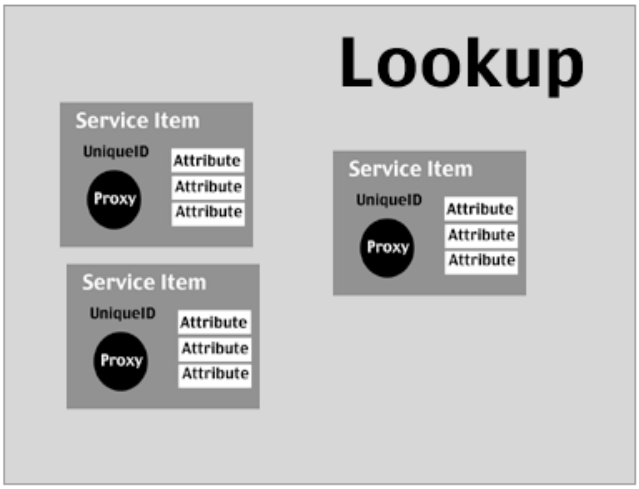
ServiceInfo Information about the service such as serial number, model name, manufacturer. SC.

ServiceType Human-readable description of the service, SC.

Status This attribute may be used by some services to indicate whether they are currently used, or to signal any special state. For example, a printer might have a ”paper jam” status.

Programmers can define other attributes, and Sun also adds new standard attributes with each new release of Jini.

Unique ID: An important property associated with a service is its unique ID. This ID is used to distinguish the service when it appears in different lookups. That means that a given service ID always refers to the same service on the network. It is returned by the first lookup met during the registration, and the service must keep it as long as at least one of its leases is maintained. Obviously, once a service has an ID, it will use it when registering with all the lookups, making sure that there is no ubiquity over the network. Problems may appear only when two TCP/IP multicast spaces join into a bigger one, though probability is really small (it’s actually lower than IP addresses conflicts). So, it is possible to make the assumption that a given ID only refers to one service in a community, and that a service in a community has only one ID. It is important to notice that a given service might change its ID after being completely disconnected from the network, even if well-behaved services save their ID to a file so that when they recover, they can come back up with the same ID.



3.2.3 Client request

At detailed in the previous definitions, we have three different things stored in the lookup that are related with our service:

- A unique ID.
- The Jini proxy.
- Some attributes.

Clients will use one or more of these things to find the service they want to use. The process is quite simple: the client builds and sends to the lookup an object implementing an interface called net.jini.core.lookup.ServiceTemplate. This object is very similar to the ServiceItem object stored in the lookup. The lookup will then find all the matching services and return one or more proxies. That enables the client to ask for a service with the following properties:

- A given ID or
- A service whose proxy is an instance of a given class (set of classes) and/or whose proxy implements a set of interfaces and/or
- A service that has one or more given attributes.

In a few words, the client must know something about either the type of the proxy, or about the attributes associated with the service. The service ID request is defined to make it possible to access again a service that had previously been used without keeping its proxy in memory.

3.2.4 Implementation overview

It's worth notice first that the word lookup refers to the lookup service as well as the things a service may do with a lookup service.

The things this lookup service do are to listen to all kinds of discovery calls related with the groups it supports, and to correctly answer to these calls. As a lookup is also a normal service, it should first do everything a service does (register with other lookups, renew its leases, ...).

Moreover, it should perform MCAP when joining a community to announce its presence to all the services of its groups.

Quite evidently, a lookup has to be able to receive, store and provide the ServiceItems mentioned in section 3.2.2. To receive the service data, it has to implement the join protocol. If a service tries to register without specifing its ID, the lookup has to provide a unique ID to the service. To provide the requested proxies, the lookup must be able to scan its database to find the matching proxies, attributes or ID's.

Furthermore, to avoid storing useless data, it should grant leases to the services that store their proxies in its database, and clean up resources whenever a service fails in renewing its lease.

Finally, a well-behaved lookup should be able to use remote events to notify clients that wish when a specific service appears over the network.

Fortunately, most of these tasks may be performed by Sun packages, making it quite easy for the programmer to build a good lookup service. Nevertheless, some advanced issues such as federating lookups are up to the programmer.

3.3 Leasing

3.3.1 Concept

As mentioned in section 1.3, a lease is an amount of time a service grants to its clients to use its resources. It is a very useful tool Jini uses to achieve self-healing.

In general, a client must renew its lease as long as it wants to use a service, and should unregister when disconnecting. The great advantage with leases is that when a client crashes or is disconnected, it isn't able to renew its lease any more, and the service realizes that the client is not connected any more.

Very often, the actual objective when dealing with leases is to check that a client of the service is still "alive"... The most common use of leases is to ensure that whenever a client crashes, all the related resources stored by the service are erased. This allows

services to avoid accumulation of stale data, and reduces therefore the number of network errors due to outdated references to services.

Another benefit of leases might be to avoid undesired use of the service. A service can deny a request for a lease or for a lease renewal. It might be possible to create services that automatically get rid of undesirable clients... Leases are actually a very powerful concept, that may be used in a wide range of situation.

As seen in section 2.2.4, a service may not have enough computational abilities to run an embedded JVM. To perform leasing, Jini allows services to use a delegate to renew their leases as well as to grant the leases they might have to issue to their clients. As before, it is up to the developer to find a way for the delegate to check that the service it deserves is still operating correctly.

One of the most important issue about network reliability is to determine correctly the duration of the leases a service should grant to its clients. The shorter the lease, the more reliable is the network. The longer the lease, the faster is the network. The compromise is therefore between reliability and performances.

3.3.2 Which services should use leasing?

In a few words, any service that maintains data related to a client should use leasing. A trivial example is the lookup service that'll issue leases to any registered service. The leases issued by the lookup should not be too long as the network's robustness depends directly on lookup's data integrity and correctness. On the other hand, all the services have to register, so that too short leases will quickly slow down the whole network. In general, lookup leases should last between 1 and 15 minutes. In some special cases, lookup leases might be even shorter, but that is not common.

For other services, leases duration may vary widely depending on many parameters. Some services might issue leases of a few seconds, while some other service that have a huge storage capacity might grant leases for several months! The only limit is given by networks infrastructure and performance as a lease cannot be renewed if the duration is shorter than the transmission time!

It is also possible to imagine other uses of leases, for example to deal easily with some security issues or to avoid services overflows when too many clients are trying to access simultaneously the services resources. To summarise, leases may be used in many situations, but primary users are the services that store client-related data.

3.3.3 Implementation overview

In Jini, leases are granted after a short negotiation between the client and the service about lease duration. As in real life, the grantor has the final word. The policy is really

trivial: the consumer asks for an amount of time, and the grantor decides whether it issues a lease or not. If the grantor decides to grant a lease, its duration has to be shorter than or equal to the requested duration. The client must then renew its lease as long as necessary, but it is still up to the service to decide whether the client is granted the renewal and if yes, for how long this renewal is granted.

As mentioned in the previous paragraphs, a lease must be short enough to ensure failure detections. But if the duration is too short, the network will be overflowed by lease renewal messages. Requirements may vary very widely depending on the application, and that is probably why Jini imposes only few operational restrictions. The only thing on which the client and service have to agree is the lease definition. Jini provides the solution through the net.jini.core.lease.Lease interface that must be implemented by the lease object the service sends to the client, so that the client can deal with any lease it receives. No assumption can be made about how the service determines the lease duration, except that the granted lease cannot be longer than the request the client issued.

To give an example, a simple decision algorithm might be that the service grants leases of any duration shorter than an upper bound. When a client sends a request for a longer lease, it is only granted a lease of the maximal duration. In some situations, more powerful and appropriate algorithms may be used to determine leases duration, but very simple algorithms are the most widely used.

A few other things are defined by Jini. The net.jini.core.lease.LeaseMap interface provides a standard for grouping several leases issued by the same grantor to renew them together. This is the case when a service issues several leases to the same client (for example for different subservices), or when several services use the same delegate to grant their leases.

As only these few features are actually defined by Jini, Sun Microsystems provide many classes that can be used to deal easily with leasing. These classes are part of the com.sun.jini package, and are therefore property of Sun, but can be used by programmers for development purposes. The most widely used of these classes are the classes of com.sun.jini.lease.landlord package that define a basic implementation of a lease and a renewal protocol.

3.4 Remote events

3.4.1 Motivations and concept

On a network, services often require to be notified when interesting things happen in the system. There are plenty of examples, from the program that wishes to print a file and that expect to be notified when a printer is available to the coffee machine that needs to detect that my alarm clock is awaking me to prepare my first cup of the day!

More seriously, the need for a system allowing notification of events on a network is obvious. The concept of event is a well-known solution to this problem. The following paragraphs briefly give a short description of this concept, which is actually widely used by many languages including Java.

Basically, an event is piece of data meaning that something happened and that may contain information about what happened. It is called event quite logically, as it always corresponds to a change in the environment, which is the actual definition of a real event.

As in real life, a computing environment is constantly evoluting, and events consequently happen continually. It is therefore necessary to what are the events we are interested in. That's why any program that wishes to be notified about a specific type of events has to explicitly tell the program that issues this type of events about its will to be called whenever an interesting thing happen.

Once a new event has been created, it is thrown. That means that it is sent to all the programs that are listening to its issuer. Then, the listeners may deal with the events they receive and react appropriately to the changes of the environment.

3.4.2 Possible requirements

From one application to another, the system requirements in event delivery may differ very widely. In some cases, it will be absolutely necessary that any event issued is sent to the listeners (for example when two chess players use computers to play, the event of playing must be received by the other participant). In other situations, it might be much more important to be able to order the events issued by several sources (for example, a server that stores bookings for a concert has to order the requests). We might also need the events to be delivered not later than a given time after they were issued (for example, a service might wish to notify that it'll be busy for the next five minutes. Once it is back, delivery of the event is useless).

The consequence is that there is no way to define a type of events for each specific need. Moreover, as listeners have to register with the issuers, they need to know the definition of the events they are interested in. Another consideration is that on a network, the costs of delivery of events are order of magnitude bigger than in the local case. Distributed systems should therefore be architected to generate as few events as possible . Both these reasons encouraged Jini conceptors to define the so-called remote events used by Jini, that are actually a very light and general type of events.

3.4.3 Implementation overview

As mentioned in the previous paragraphs, Jini provides an event-programming model based on remote events. This type of events is defined by the net.jini.core.event.RemoteEvent class that must superclass any event thrown over a Jini network.

This class declares four variables associated with any remote event. These four information are:

1. The event ID, which is key associated with all the events a specific service issues (this key might not be unique).
2. The sequence number, which is a number used to order the events a specific service throws. The greater sequence number, the later the event has been thrown. This number might also be used to determine whether all the messages until the last received event have been delivered or not.
3. The event source, which is actually the piece of software that issued the event.
4. A handback object, which is supplied by a specific listener during registration and that is attached to any event sent by the service to this client.

In most cases, the events will subclass RemoteEvent to provide more information, for example about what happened... Jini also defines the net.jini.core.event.EventRegistration class that is used to facilitate the handling of events on the client side and the net.jini.core.event.RemoteEventListener interface that must be implemented by any client that wants to receive remote events.

That is a really simple system, not very expressive. That might seem to be a bit too light to meet the needs for information on the network, but it is always possible to subclass RemoteEvent, and the usage of a unique structure make it possible to create generic delegates. These generic delegates are pieces of software that can use, store, modify and/or forward remote events without knowing anything about their type except that it extends RemoteEvent.

One very interesting feature of the generic delegates is that they may form a pipeline. That means that an event thrown by a service may go through several delegates before it reaches its destinations. Any of these delegates might modify it, or store it, or even order it.

The event model strongly uses leasing to ensure event issuers do not spend their resources trying to deliver events to crashed clients. Especially when the consequence of non-delivery is the suspension of the service, the lease will be very short to avoid at most interruptions of work.

Chapter 4

Advanced Jini

4.1 Transactions

4.1.1 Motivations and concept

One of the advanced features Jini provides are transactions. A transaction is an operation involving several participant services or clients, and that has to be treated as an atomic operation. That actually means that either all the tasks of the operation succeed, or none of them. These two possible conclusions of the transaction are respectively known as the committed state and the aborted state. To summarize, the aim of transactions is to avoid any type of partial failure that could lead to a partly-only completed operation.

The operations grouped into a transaction have to be treated as one atomic operation. As they cannot be physically treated simultaneously, transactions have to provide a way to logically simulate atomicity. We will see in the implementation overview how Jini deals with this difficulty, but I'd like to underline already now that the changes the operations involve have to remain hidden as long as the transaction isn't committed. That means that original data cannot be directly modified until all the operations complete. Transactions are therefore invisible as long as they aren't committed.

Transactions are very complex, as many problems may trouble the normal process of the operations. Some particular difficulties appear when simultaneous transactions are performed. Some data should not be modified by several transactions at the same time. For example, a bank account should not be decreased simultaneously as the total debited amount could exceed the available money. So it is often useful to ”queue” the transactions using semaphores to allow data to be modified.

As this report is actually a global presentation of Jini, I'll limitate myself to the simplest situations, which are fortunately the most common. Special cases of transactions' use are beyond the scope of this report, and might be the subject of another report.

4.1.2 Implementation overview

As mentioned in the previous paragraphs, a transaction is a group of operations involving several parties. In Jini, there are three distinct type of parties in each transaction:

- The client, that decides which operations have to be grouped, and that initiates the transaction. While any party can abort it, only the client can commit the transaction.
- The manager, that is intended to oversee the transaction. It is up to the manager to make sure that consistency is maintained, and to control the whole processing of the transaction.
- The participants, that are actually the services that perform the operations the client wishes to group together. Services may or may not be designed to deal with transactions, and it isn't be possible to group services that aren't transaction-enabled into a transaction. The rules transaction-enabled services have to follow are clearly defined by Jini (through some interfaces), so that it is possible to use transactions even without prior knowledge about services implementations.

The protocol is a quite simple step by step process:

1. The client that wishes to initiate a transaction must first find a transaction manager. This manager is actually a Jini service, and is therefore registered with the lookup, so that it can easily be found and called.
2. Once the client has called the manager, it is returned an object implementing the net.jini.core.transaction.Transaction interface. This object contains a unique ID that is used to isolate and distinguish the transaction.
3. The client will then send this object to all the participants everytime it calls them. The participants are therefore aware that any change the make is conditioned by the commitment of the transaction, and that any change must be kept invisible to any client that does not participate in the transaction.
4. Whenever an operation cannot be completed, the client can request that the transaction be aborted. If it doesn't, all the operations complete.
5. Once all the operations are completed, all the participants are told to prepare to commit. They can either agree, or drop out of the transaction. The fact of accepting actually means that the participant guarantees that transaction data will be committed or aborted in any case. The fact of dropping out means that the service will clean any transaction-related data and won't be asked neither to commit nor to abort.
6. Either all the participant agreed, or not. In the first case, the manager will ask all the participants to commit the changes. In the second situation, one or more participants drop out, and the manager will ask all the participants to abort the transaction. The manager eventually checks that all the participants have actually committed or aborted the transaction.

The two last steps are known as the two-phase commit protocol. To conclude about how Jini implements transactions, I'll notice that Jini's core packages provide a few interfaces that have to be implemented by the parties of a transaction. How they are implemented is up to the programmer, and some guarantees may not be respected by other people's implementations.

4.1.3 Notes

Transactions are quite complex, and not always necessary. When it is really worth using transactions, it is often better to chose a simple model. One good solution is to use the classes provided by Sun Microsystems in their com.sun.jini package. In particular, they provide a simple but easily usable transaction manager.

To conclude with transactions, there are two facts I have to mention:

- Jini supports a concept called nested transaction. A nested transaction is a transaction initiated by another transaction. When this particular type of transactions are aborted, the parent transaction remains unaffected, while when they are committed, the data changes they involve aren't made before the parent transaction completes.
- Another remark is about guarantees. The assumption a service can commit or abort a transaction in any case is not possible. In fact, when a service agrees to commit, it often keeps a representation of its state on a persistent storage, so that if crashes, it can complete the request once it is restarted. But, for example, a device could possibly be totally destroyed just after it accepted to commit a transaction, and the guarantee is therefore not absolute.

4.2 JavaSpaces

4.2.1 What is JavaSpaces?

A storage system for Jini JavaSpaces is a service that ships with Sun's implementation of Jini, and that is intended to provide a storage system for Jini services. However, JavaSpaces provides a service that is very different from what a common database or an usual filesystem would provide. Each of the three main differences is the subject of a subsection. These crucial differences are:

- Data that may be stored.
- The way stored data is referenced.
- The transience or persistence of the provided storage.

As JavaSpaces is a common Jini service, it has to register with the reachable lookups, and can consequently be found through a single classname-based request to the lookup. Moreover, JavaSpaces implements a complete support of transactions.

An object-centered storage system While databases and filesystems usually accept to store almost any type of data, JavaSpaces only stores Java objects. Stored data is therefore strongly typed, and as it is Java code, it is moreover mobile (i.e. it can be used on a wide range of devices). These properties define what is known in Jini literature as an object-centered storage system.

The particular object-centered storage system JavaSpaces provides manages some storage capacity. The capacity provided by this service is often called a space, i.e. some finite place that can be used to store objects.

In practice, the process of storing a Java object on a remote service requires this object to be serialized. This fact involves that anything stored on a space is actually a Java serialized object. What this object represents, whether it is unique or not, whether it is actually information or executable code is irrevelant.

In the next section about attribute-based search, the way Jini clients find stored data they are looking for is clearly detailed. The important thing here is that, as Jini proxies are, the Java objects spaces can store have to be associated with attributes in an object implementing the net.jini.core.entry.Entry interface. The stored object are therefore referred to as entries.

To summarize, the only requirements an entry has to fulfil to be storable on a space are to be a serializable Java object implementing the Entry interface. The service that wishes to store this object first has to find a JavaSpaces service via a lookup, and can then serialize and send its object to the space. The details of how it actually performs this operation are explained in the implementation overview.

Entries and attribute-based search Databases and filesystems normally reference the stored entities either by some identifier, or by the unique position where they are actually stored. In almost any case, the storage system provides a way to access a specific object, making it possible for the data consumers to easily access what they are looking for. Very often, the stored entities (distinct pieces of code) either have a name (for example files), or a unique key. That means that any stored entity is distinct and unique.

On the other hand, JavaSpaces uses the same system the lookup uses to reference stored data. As for the lookup, the search is based on attributes the client provides to the storage service. That means that a client that wishes to find an entry has to provide the space a template. The whole process is defined in the section about entries, attributes and templates, and the only difference is that JavaSpaces do not support multiple-templates requests and can return only one entry to the client.

In many cases, several entries will match the request, and the JavaSpaces service will return randomly one of the matching entries. Nevertheless, as illustrated in the programming tips section, there are easy ways to obtain several or all the matching

entries. The important thing to remember is that there is no reason the same request would give the same result twice.

As lookup services do, spaces accept client requests to be notified when a new entry matching a template is stored. This is possible even if a stored service does already match the template.

To summarize, requests on spaces follow mostly the same rules they follow on lookups, and it is also possible for a client to ask to be notified when matching entries appear.

Storage properties While filesystems and databases are usually intended and designed to provide persistent storage, JavaSpaces may be transient as well as persistent. Sun's JavaSpaces ships with two implementations that are respectively the transient one and the persistent one.

The transient implementation of JavaSpaces provides the community what is called a transient space, as the persistent one provides a persistent space.

The only difference is that when a transient space crashes or restart, stored data is lost, while on a persistent space, data can be recovered as it is logged on a harddisk.

In both implementations, a very important thing to underline is that the storage capacity is leased. That means that only "alive" services can store objects on a space. Whenever an entry lease is not renewed, the space can erase the entry. There is actually no way to make sure that the entries we store on a space will still be there later. Therefore, any data that shouldn't be erased should be copied locally when stored on a space.

To conclude this little note about persistence, it is worth notice that databases and spaces are very different, and that they can be used together. Their conjugate use can provide the community an appropriate storage for almost any need.

Possible usage JavaSpaces is a powerful tool for storing and sharing data. The way spaces can be used are numerous, and these are the most common:

- Services that do not have sufficient local storage capacities can store data on JavaSpaces.
- Services that want to publish data can just store their entries on spaces.
- Services that need to transfer data in asynchronous way (i.e. the sender wants to send at time a, and the receiver wants to receive at time b) may use spaces as a temporary storage for the data they want to transfer.

- Services that work simultaneously with the same information can use a space as a common reference for updated data.
- Services that cooperate may use a space as a whiteboard to communicate indirectly, avoiding by the way to download each others proxies.

As the set of possible usages is really huge, and as spaces are really polyvalent, it is up to the programmer to imagine how this service could help him in his particular situation. And it is really likely that JavaSpaces will help him everytime he needs some storage abilities.

4.2.2 Implementation overview

The JavaSpaces interface (net.jini.space.JavaSpace) is actually defined by Jini specs. However, the implementations I described in the previous section are proprietary. That means that it is possible to implement other type of JavaSpaces. For example, it would be possible to create a service that would provide a non-leased storage space (even if that would endanger self-healing and reduce robustness).

The JavaSpace interface defines seven methods that are intended to provide four functions and an optimization. The methods are presented in the following next paragraphs that present the four functions:

1. Write an entry
2. Read an entry
3. Read and delete an entry (take an entry)
4. Ask to be notified when a matching entry appears.

Write an entry To write an entry, a client has to call the write method and to provide the entry object, as well as the required lease duration. Moreover, the write method can take a transaction as parameter, so that the entry is not made visible as long as the transaction is not committed. Obviously, if the transaction is aborted, the entry is deleted.

Read an entry To read an entry, a client may use two methods: the read and the readIfExists methods. The client has to provide a template entry, as well as a timeout duration. If an object exists and is visible, both methods will return it to the client. If no object exist, the read method will wait until a matching object appears or until the timeout expires. In case the timeout expires before an object matches the template, a null object is returned. On the other hand, the readIfExists method will return null immediately unless an invisible matching object already exists. In this case too, if the timeout expires before the object is visible, the method will return a null object. To conclude with reading, an optional transaction parameter can be passed to both the

methods. The consequence is that the client will see as visible all the visible objects plus the objects created by the given transaction.

Take an entry The fact of taking an entry is defined as reading and deleting it. The methods are therefore quite the same as the reading methods, and have the same parameters. They are called take and takeIfPossible. The only detail to notice is the consequence of passing a transaction as a parameter, as it will mean that the space will have to wait until the transaction is committed to delete the entry.

Ask to be notified The request to be notified is not surprisingly performed by the notify method. The five parameters of this method are respectively the entry template, an optional transaction object, a remote event listener that will receive the events, a required lease duration, and a handback marshalled object (definition and details in the remote events section). This actually quite a standard notification request, and the only subtle things are about timeout. Basically, the notification request will last as long as the lease is renewed. However, if a transaction is passed as a parameter, notification will end at latest when the transaction completes.

The last method called snapshot is used to optimize the use of spaces. As it often happens that a client iterates calls with the same entry template, and as this entry might be heavy, the snapshot method allows a client to store an entry on the space. The method will return another lighter entry, that will be used instead of the original one when iterating calls. This returned entry may differ from a space to another, so that the client has to get a new entry object for any space it is dealing with.

4.2.3 Programming tips

As JavaSpaces only defines really few methods, some features might seem to be missing. In fact, almost any common operation on data can be performed with a sequence of calls of the seven methods the interface defines.

A trivial example is the method to get all the entries that match a template:

1. Create a new transaction object with a transaction manager (as described in the section about transactions)
2. Iterate the take method with the transaction as long as no null is returned
3. Abort the transaction to avoid the entries be deleted

Other simple ideas may extend the possible operations. For example, entries could be referenced specifically by adding them unique attributes. The best solution when trying to implement a new functionality is to ask a web-based forum if anyone has already done it!

4.3 Administration tools

4.3.1 Motivation

A Jini service doesn't need much administration to work correctly. However, almost any Jini service still needs some minimal user-interventions. Moreover, many Jini services need to be automatically administered and controlled by other services. Therefore, there must be a way to access services settings remotely, and this is actually called administration in Jini.

For example, it might be really useful to be able to start or stop service remotely. The controller of a central heating of a theatre should for instance be administrable from the cash desk.

The next subsection gives a quick overview of how Jini deals with administrative issues.

4.3.2 Overview

As a really wide range of services may use Jini, it would be really difficult to define a finite set of common needs for administration. Hence, Jini does not define how services should administrated. However, to give "foreign" services (that have almost no prior knowledge of each other) a chance to administrate each other, Jini provides an administration framework.

This framework comes with a few almost-always useful administrative functionalities and lets services define additional administrative features. The functionalities are made accessible to the clients via common administration interfaces. The administration interfaces that come with Jini are:

net.jini.lookup.DiscoveryAdmin This interface allows clients to control a lookup service. That means that the lookup allows some or all clients to modify the groups it participates in, as well as a few other parameters.

net.jini.admin.JoinAdmin This interface allows clients to control the participation of a service in the join protocol by specifying the groups, attributes and lookups that the service should use.

com.sun.jini.adminDestroyAdmin This interface allows clients to actually stop the service.

com.sun.jini.admin.StorageLocationAdmin This last interface allows clients to control where the service stores persistent data.

Each of these interfaces define a few methods the administrable service has to implement. Then, a service can implement the net.jini.admin.Administrable interface, to tell "foreign" services that it has been designed to be remotely administrated.

Administration is not always useful when developing. However, it should not be forgotten, as it is really useful when it comes to production environments, as it makes it much easier to modify the settings of the services.

Chapter 5

Presentation conclusion

5.1 A few words to conclude

As shown in this first part, Jini provides a complete model to build new robust and reliable networks. Moreover, as often with Sun products, Jini benefits of a strong and enthusiastic community of developers, that really want Jini to impose itself as THE solution to build any type of distributed systems. ini might therefore become very popular, as Java did, and is consequently worth being studied carefully.

5.2 My opinion about the model

I think that Jini provides some interesting solutions to the common networking problems. Especially, some concepts such as discovery, lookup registration or leasing seem very useful to me. Advanced features such as transactions are as useful in Jini as in other environments, and some others, as JavaSpaces, have a great potential to change the way we build networks.

On the other hand, all this is quite heavy, and that's a problem when it comes to performance. Moreover, some concepts still need to be developed and clarified, such as implementation agnostism. The way Jini references services is actually not very efficient, nor does it really allow "foreign" services to cooperate as fluently as expected. Especially when dealing with hardware, Jini doesn't provide enough common interfaces yet to define the classical devices that are connected to the network. All this is evoluting, and some near-future versions of Jini will fix these little insufficiencies.

Nevertheless, the Jini model is clearly a potentially revolutionary model, that opens new horizons in building networks. Many concepts are very innovative ideas, that will certainly change the approach we have in creating distributed systems. As Java did change the way programmers work, Jini will certainly help networked systems conceptors to build better product.

Anyway, such a presentation is only a presentation and doesn't replace an actual experience in programming with Jini, which is the subject of the next part of this report.

Part II

A practical example

Chapter 6

Description

6.1 A need for practice

When working with new technologies such as Jini, it is often difficult to know whether the model is efficient practically. The best way to evaluate the real potential a technology offers is certainly to experiment its use in practice.

The main objective of the second part of this report is to show a simple and basic example of the reality of Jini, and to give the reader a first idea about Jini programming,

6.2 A trivial event orderer

6.2.1 Why this choice?

Making a sensible choice for the example to implement and describe isn't easy. The example should illustrate at least all the fundamental concepts of Jini, and use at most the provided facilities to build a service. Moreover, as the classical HelloWorld examples are terribly boring, this example had to present at least a little interest. Eventually, the main requirement such an example has to fulfils is to provide a basis for future works.

All these considerations leaded to the choice of a trivial event orderer.

The problem of delivery order is really complex, and there is no simple solution to order events over a network, as there is no central reliable time by default. Many algorithms have been developed and are actually used to ensure that events are delivered following a unique order over a network.

This is far from this little semester project report about Jini technology, and the reader will not find here anything else than a very trivial system to associate positions to events. Once again, this example is intended to show how to use the classes provided

by Sun's implementation of Jini. However, the topic of in-order delivery is still much more interesting than the usual HelloWorld's.

6.2.2 Structure overview

The trivial event orderer that is presented in this part of the report has to use at most Jini facilities, and will obviously be implemented as a service.

This service has to act after the event is sent and before it is delivered, which means that our service will play an intermediate rule between the sender (the event generator) and the receiver (event listener). The objective is to find a way to order the events, so that any listener has the same sequence of events.

The whole system will be composed of several different players:

- The lookup(s) service(s) of the community.
- The event generator(s) that create(s) and send(s) the events.
- The event listener(s) that want(s) to be notified when some event are sent.
- The event orderer(s) that should provide the community a shared common sequence of events.

The example will show the mechanisms Jini provides to these players to interact with each other.

6.2.3 The covered topics

The event orderer service (EOS) is a standard Jini service, and will therefore register with the lookup and provide a proxy for its clients. This part will illustrate the discovery process as well as the practice of the main features the lookup concept defines.

Moreover, EOS has to register with the generators in order to receive the events they send. This will illustrate leasing registration from the client side, with a practical implementation of multiple leases renewal.

Furthermore, as the listeners want to be notified whenever an event is sent, our service will have to issue leases to these listeners in order to notify them about new events. This will illustrate leasing from the landlord viewpoint.

Eventually, as it mainly deals with remote events, our example will widely cover this topic.

To summarize, our trivial example uses all the fundamental concepts of Jini, which is exactly what it was intended to.

6.2.4 The model

A very simple solution is to associate a position to these events, and to send them further. With this system, even if the events do not arrive in the same order everywhere, the position gives the listeners an easy way to reconstruct the common sequence.

So, the process is really trivial:

1. The generators send their events.
2. The EOS receives the events.
3. It associates a position to each event.
4. It sends each event to all the registered listeners.

The position associated with each event a single player sends is usually referred to as sequence number. As this is the choice we made to order the events, the position will be called sequence number. This number should follow some rules to have an actual meaning:

- This sequence number should obviously be unique (property called Idempotency, and required by Jini spec).
- If we want our listeners to be able to reconstruct the sequence, we have to specify a policy such as ”new sequence numbers are always bigger than older” (property called Strict Ordering, required by the Jini spec).
- Eventually, if we want our listeners to notice whether they received the complete set of events that have been delivered, sequence numbers should not be skipped (property known as Full Ordering, optional in Jini).

There are many possible properties an EOS could implement. For example, it could check that a given event hasn't been forwarded twice, or it could decide that events must be delivered at any cost, and keep trying to deliver some events as long as the lease of some crashed listener is cancelled.

The EOS that is implemented by the example doesn't implement the above properties, but uses Full Ordering, and offers therefore a few guarantees, that are listed in the next section.

6.2.5 The guarantees of the model

The EOS that is implemented has the following properties:

- It uses Full Ordering (and obviously, Strict Ordering and Idempotency).
- It doesn't try to deliver events more than once to each listener.
- It doesn't check that an event is forwarded only once.

- It doesn't destruct any information contained by the forwarded event.

The EOS can therefore offer some guarantees:

- The event sequence can be reconstructed by the listeners.
- The listeners can ensure they received all the forwarded events, and can notice if it is not the case.
- The listeners can delete the events forwarded more than once by comparing the ID's of the forwarded events.

These guarantees are very interesting, and this is actually a good trivial model. However, there are many useful guarantees that cannot be made with such a simple scheme. The two main problems are:

- The undelivered events cannot be obtained by the listeners.
- The system is not reliable, as the EOS is single point of failure!

Nevertheless, this model will be retained for this semester project, and a better implementation of an EOS could be the objective of another work.

Chapter 7

The components

7.1 Overview

The various components that our event orderer example will use are:

- An event generator service.
- An event listener.
- The actual EOS.

To make things a bit more nice to read and to experiment, the events will not be ”empty”. Each event will actually contain an integer representing a person, Alice, Bob, Cecily or David. Imagine for example that an event containing Alice is sent by a service PhotoAlbum whenever a user of the service clicks on her picture.

So, there are four possible events, and they are all ”people events”.

One or more generators will send people events, and some listeners would like to receive these events with the same sequence everywhere.

To achieve this, the EOS will associate a sequence number to each event between the generator(s) and the listeners.

The technique that EOS will use is that it will listen for people events, receive the people events, nest them into ”ordered events” with a sequence number, and send these ordered events to the listeners. The listeners will then get the nested event and the associated position in the sequence.

Each component of our example will correspond to one or more Java class(es) that are grouped in the jiniproject package.

7.2 A simple Jini service: Abstract Class BasicService

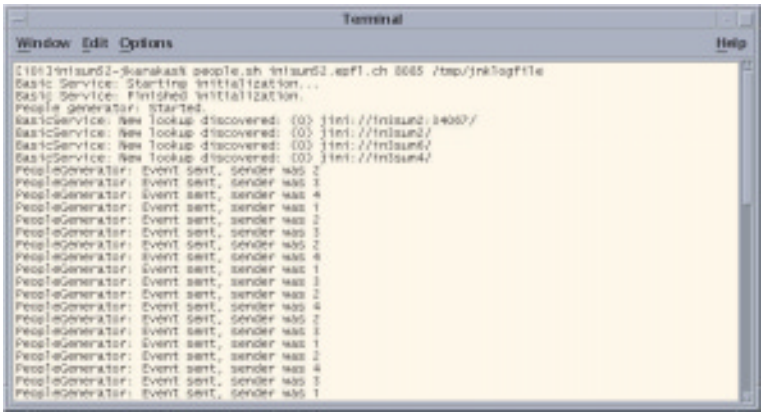
In order to increase re-usability of the code, the first class that we will define is a superclass for all simple services.

This service do the following things for its subclasses:

1. Discovers all the lookups that run on the same IP multicast network as the device running the service using MCRP.
2. Registers with all these lookups by providing a Jini proxy its child has to create.
3. Keeps listening for appearing lookups, and registers with the news ones.
4. Stores to persistent state its unique ID returned by the first lookup discovered.
5. Stores to persistent state the list of the active lookups in order to recover faster by using UCDP.
6. Renews all the leases granted by all the lookups.

This service is therefore a fully-fledged Jini service, that can be used as a superclass for many basic services. It uses the new Jini 1.1 final facilities, and will not work with the alpha version of Jini 1.1.

7.3 An event generator: Class PeopleGenerator



This simple subclass of BasicService sends periodical events for each person (Alice, Bob, Cecily and David), and allows listeners to register separately for each person.

The interesting features of this service are that:

1. It shows an example of a simple service proxy.
2. It illustrates event generation and delivery.

3. It shows how to manage multiple and separate leasing (listeners register separately for each person).

The way this service decides to send events (i.e. periodically, with different periods for each person), is not very interesting. It is actually intended to generate events to test the EOS, not less, not more.

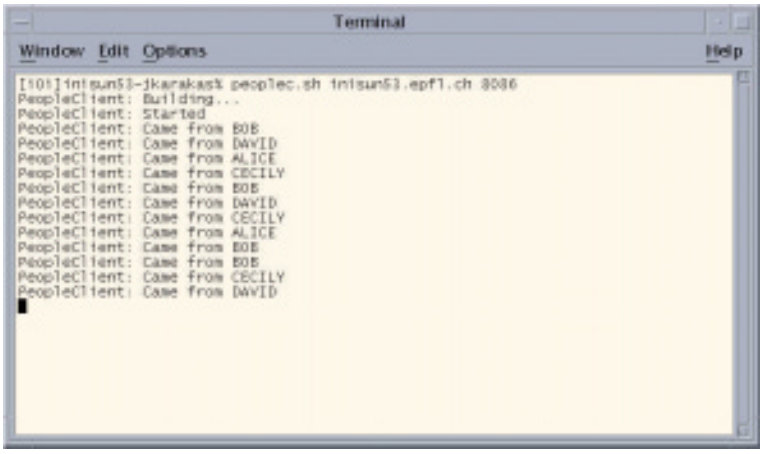
7.4 A landlord for event registrations: Class SimpleEvtRegLandlord

This class is used by PeopleGenerator to manage event registration requests. An instance of this landlord can grant leases for a single event type. It is an implementation of a Landlord that may be used by any service that wishes to grant leases to event listeners.

7.5 A representation of a registration: Class Registration

This class only stores all data relative to a specific registered client (i.e. a listener in this case).

7.6 A listener: Class PeopleClient



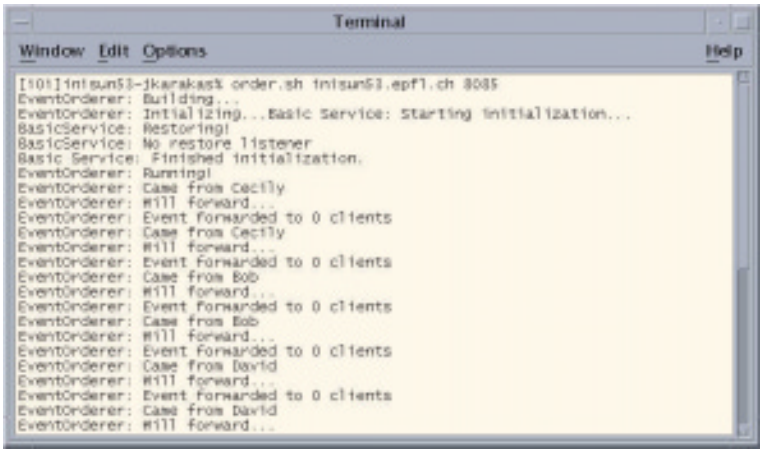
This class is a simple listener of people events, that doesn't listen for ordered events. It shows how to:

1. Ask a lookup for a specific proxy by passing the classname of an interface the proxy implements.

2. Register with an event generator.

3. Receive the events.

7.7 The actual EOS: Class EventOrderer

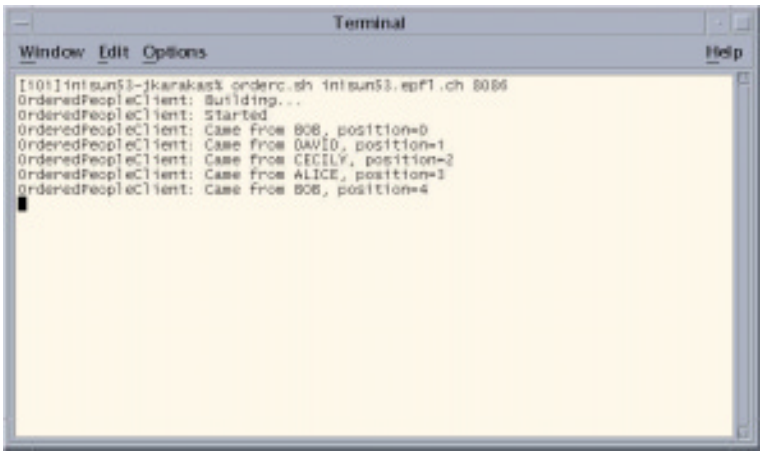


This class is the heart of the example. It receives people events, nest them into ordered events, and associates a sequence number with each of these events.

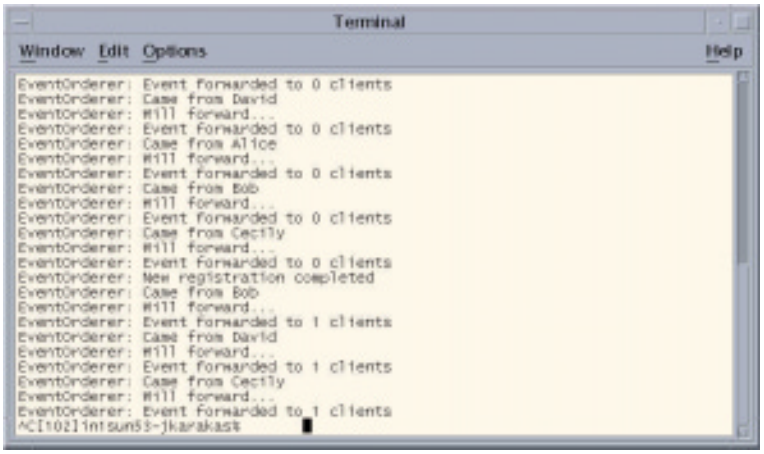
This service performs therefore these three distinct operations:

- It receives people events as PeopleClient did.
- It sends ordered event as PeopleGenerator did.
- It creates a new type of remote events that embeds a people event as well as a sequence number. This third function is new, and illustrates the great potential remote events offer.

7.8 A client for EOS: Class OrderedPeopleClient



This class is similar to PeopleClient excepted that it has to extract the nested event before it can use it.



7.9 Compiling and running these examples

To compile the files, simply use javac and compile the whole package at once. The classpath must contain the net.jini, net.jini.core and sun.com.jini packages, as well as all the Java 1.2 classes. However, as all the proxies of this example extend java.rmi.UnicastRemoteObject, they all need to be ”rmic”-ed.

As our client are quite simple, they do not ask the lookup to be notified when a new generator appears. So, to run our example, we need to first start PeopleGenerator, then EventOrderer, and finally OrderedPeopleClient and/or PeopleClient.

7.10 Example conclusion

This was a simple, but complete example, that can help the reader to understand the mechanisms Jini defines. Moreover, both the BasicService and the SimpleEvtReg-Landlord are general enough to be re-used without any change.

Obviously, it would be possible to implement a much better EOS. In particular, both the problems listed at the end of pt 6.2.5of may be solved, at least partly, by more complex schemes.

The first one could be solved by a widely redundant use of services such as event mailboxes, that would store events to persistent storage. These mailboxes could either store only the events that couldn't be delivered to all listeners, or all the events. In the second case, a new listener could even access past events.

The second problem could be partly solved by a system using several EOS. They should then have to agree on which EOS decides for the others. For instance, the policy could be that the EOS on the machine that has the smallest IP address defines the unique event order. If this machine crashes, ther will still be another EOS that has the smallest IP...

Chapter 8

Project conclusions

8.1 My point of view about Jini

I actually don't have any experience in building networks with technologies such as Jini, so my opinion is not a reference. However, I have worked with Jini for the last months, and I could appreciate both the efficiency of the model's implementation and the relation between the model and the implementation.

Concretely, the implementation doesn't match the model on several points. Device agnostism is not achieved, and some problems still happen with Jini. The implementation is sometimes quite heavy, and Jini will never be optimal for performance (as Java isn't).

On the other hand, once a service works correctly, it is long-lived. For example, I forgot to stop a lookup on a machine in IN3, and found it again two weeks later, still running correctly. Jini is also quite easy to use. Moreover, as it is based on Java 1.2, it is possible to use it with almost any type of personal computers. And last but not least, there are many great people that work to impose Jini as a new standard.

I think Jini is a potentially interesting technology. If provides useful tools to build good networks, and its model is clever as well, and I strongly recommend to evaluate its possible usage when designing new distributed systems.

8.2 My opinion about this project

I am really glad to have discovered Jini technology, and it was very interesting to work ”freely”. Obviously, it hasn't been easy as this project started at point zero. As Jini is a young technology, only few documentation is available. Moreover, the transition between Jini 1.0 and 1.1 was quite brutal, and all the books that were available three

months ago are already outdated.

However, I think I have been lucky to have the opportunity to work with such a technology. I have also learnt a lot about networking, as well as some interesting programming techniques. I could work with RMI, java 1.2, and also discovered the wonderful tool LaTeX is.

To conclude, I hope my work will be useful for LSE projects, and would be really glad to be informed if any Jini-based project is initiated by the lab (Maybe I could help?)...

Appendix A

Source code of the example

A.1 Class BasicService: A generic Jini service

```
// ==============================
// *** Abstract Class BasicService ***
// ==============================
// Usage:
// This class defines a standard superclass for many simple services
// and provides all the basic methods for a service to re-
cover after
// a crash.
// ==============================
// Nested classes:
// -Static Class PersistentData
// -Interface RestoreListener
// ==============================
// Comments:
// This abstract class extends UnicastRemoteObject and its children
// shall therefore be "rmic"-ed before being used.
// ==============================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ==============================

package jiniproject;

// The needed packages:
import net.jini.core.lease.*;
import net.jini.core.lookup.*;
import net.jini.core.entry.*;
import net.jini.core.discovery.*;
import net.jini.lookup.*;
import net.jini.lease.*;
import net.jini.discovery.*;
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.lang.reflect.*;

// This class implements ServiceIDListener and shall there-
fore save
// its unique ID to persistent storage.
// This class also implements DiscoveryListener, and shall consequently
// register with new lookups when they appear.
public abstract class BasicService extends UnicastRemoteObject
   implements ServiceIDListener, DiscoveryListener {
```

```
   protected JoinManager joinManager;
   protected String storageLoc;
   protected File file = null;
   protected ServiceID serviceID = null;
   protected LeaseRenewalManager leaseManager;
   protected RestoreListener restoreListener = null;
   protected Object proxy = null;
   protected LookupDiscoveryManager ldmanager = null;

   // Nested class that deals with persistent storage of data.
   static class PersistentData implements Serializable {
       ServiceID serviceID;
       Entry[] attrs;
       String[] groups;
       LookupLocator[] locs;
       Object subclassData;

   // Constructor of the nested class
       PersistentData(ServiceID serviceID, Entry[] attrs, String[] groups,
                      LookupLocator[] locs, Object subclass-
Data) {
           this.serviceID = serviceID;
           this.attrs = attrs;
           this.groups = groups;
           this.locs = locs;
           this.subclassData = subclassData;
       }
   }

   // Nested interface defining the service can recover af-
ter a crash.
   public interface RestoreListener {
       public void restored(Object subclassData);
   }

   // Constructor method
   public BasicService(String storageLoc) throws RemoteEx-
ception {
       super();
       this.storageLoc = storageLoc;
       file = new File(storageLoc);
   }

   // These methods must be shadowed by subclasses as they are declared
abstract.
   // They shall return the Jini proxy of the subclass ser-
```

```
vice and the
attributes
   // of the service.
   protected abstract Object getProxy();
   protected abstract Entry[] getAttributes();

   // This method may be subclassed
   protected void shutdown() {
       System.exit(1);
   }

   // Stores the service unique ID as soon as it receives it
   public void serviceIDNotify(ServiceID id) {
       serviceID = id;
       try {
           save();
       }
   catch (IOException ex) {
           System.err.println("BasicService: Trouble saving: " +
ex.getMessage());
       }
   }

   // Called if the subclass has no data to store persistently
   protected void save() throws IOException {
       save(null);
   }

   // This method allows the storage to persistent storage of all important
   // data the service will need to recover after a crash, in-
cluding data
   // specifically needed by the subClass.
   protected void save(Object subClassData) throws IOExcep-
tion {
       PersistentData data;
       data = new PersistentData(serviceID,
                     joinManager.getAttributes(),
                                 ldmanager.getGroups(),
                                 ldmanager.getLocators(),
                                 subClassData);
       ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(file));
       out.writeObject(data);
       out.flush();
       out.close();
   }
```

```
   // This method is called whenever a new lookup is found, and will start
the
   // registration of the service with the new lookup(s).
   public void discovered(DiscoveryEvent e){
   ServiceRegistrar[] newRegs=e.getRegistrars();
   LookupLocator[] toAdd=new LookupLocator[newRegs.length];
   for (int i=0; i<newRegs.length; i++){
       try{
       toAdd[i]=newRegs[i].getLocator();
       String lookupURL=toAdd[i].toString();
       System.out.println("BasicService: New lookup discov-
ered: ("+i+")
"+lookupURL);
       }
       catch(RemoteException ex){
       System.out.println("BasicService: Remote exception, the "+i+"th
lookup couldn’t be found");
       }
   }
   ldmanager.addLocators(toAdd);
   }

   // When some lookup disappear, this method ensures they won’t be called
after
   // the next recovery
   public void discarded(DiscoveryEvent e){
   ServiceRegistrar[] oldRegs=e.getRegistrars();
   LookupLocator[] toRem=new LookupLocator[oldRegs.length];
   for (int i=0; i<oldRegs.length; i++){
       try{
       toRem[i]=oldRegs[i].getLocator();
       String lookupURL=toRem[i].toString();
       System.out.println("BasicService: Old lookup discarded: ("+i+")
jini://"+lookupURL);
       }
       catch(RemoteException ex){
       System.out.println("BasicService: Remote exception, the "+i+"th
lookup couldn’t be found");
       }
   }
   ldmanager.removeLocators(toRem);
   }

   // This method is called once, at startup, and allows the ser-
vice to
```

```
recover the
   // last state it was in before stopping.
   protected void restore() throws IOException, ClassNotFoundEx-
ception {
       ObjectInputStream in = new
           ObjectInputStream(new FileInputStream(file));
       PersistentData data =
           (PersistentData) in.readObject();

       if (data == null) {
           System.err.println("BasicService: No data in stor-
age file.");
       }
   else {
           System.out.println("BasicService: Restoring!");
           serviceID = data.serviceID;
           leaseManager = new LeaseRenewalManager();
           ldmanager = new LookupDiscoveryManager(data.groups, data.locs,
this);
           joinManager = new JoinManager(proxy, data.attrs, data.serviceID,
ldmanager, leaseManager);
           if (restoreListener != null) {
               System.out.println("BasicService: Calling restore
listener!");
               restoreListener.restored(data.subclassData);
           }
       else {
               System.out.println("BasicService: No restore listener");
           }
       }
   }

   // Subclasses have to override this to do their own initialization
behavior.
   protected void initialize() throws IOException, ClassNot-
FoundException {
   System.out.println("Basic Service: Starting initialization...");
   // Always set a security manager !!!
       System.setSecurityManager(new RMISecurityManager());
       if (this instanceof RestoreListener) {
           restoreListener = (RestoreListener) this;
       }
       proxy = getProxy();
       if (file.exists()) {
           restore();
       }
```

```
       if (joinManager == null) {
           leaseManager = new LeaseRenewalManager();
           ldmanager = new LookupDiscoveryManager(new String[]{""}, null,
this);
           joinManager = new JoinManager(proxy, getAttributes(), this,
ldmanager, leaseManager);
       }
   System.out.println("Basic Service: Finished initialization.");
   }
}
```

A.2 Class PeopleGenerator: A source of events

```
// ================================
// Class PeopleGenerator
// ================================
// Usage:
// A simple service that sends events periodically, and that
// associates a person to each of these events
// ================================
// Nested classes:
// -Class PeopleCookie
// -Static Class Registrations
// -Class PeopleEvent
// -Interface PeopleRequest
// ================================
// Comments:
// This class extends BasicService and shall therefore be
// "rmic"-ed before being used.
// ================================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ================================


package jiniproject;

import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.UnknownEventException;
import net.jini.core.entry.Entry;
import com.sun.jini.lease.landlord.LandlordLease;
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.MarshalledObject;
import java.rmi.server.UnicastRemoteObject;
import java.io.Serializable;
import java.io.IOException;
import java.util.Vector;

class PeopleCookie implements Serializable {
   //The variable used to represent the sender of the event
   int sender;
   long eventType;

   PeopleCookie(int sender, long eventType) {
       this.sender = sender;
       this.eventType = eventType;
   }
   public boolean equals(Object other) {
       if (!(other instanceof PeopleCookie))
           return false;
       PeopleCookie cookie = (PeopleCookie) other;
       return cookie.sender == sender && cookie.eventType == eventType;
   }
}

public class PeopleGenerator
       extends BasicService
       implements BasicService.RestoreListener, Runnable,
PeopleGenerator.PeopleRequest {
   // 10 minutes
   protected static final int MAX_LEASE = 1000 * 60 * 10;
   protected boolean done = false;
   protected long tickCount = 0;
   // Lists of registrants.
   protected Vector alice = new Vector();
   protected Vector bob = new Vector();
   protected Vector cecily = new Vector();
   protected Vector david = new Vector();
   // Separate landlords for each list.
   protected SimpleEvtRegLandlord aliceLandlord;
   protected SimpleEvtRegLandlord bobLandlord;
   protected SimpleEvtRegLandlord cecilyLandlord;
   protected SimpleEvtRegLandlord davidLandlord;
   protected long lastEventSeqNo = 0;
   protected LandlordLease.Factory factory = new LandlordLease.Factory();
   protected long nextEventType = 1;
```

```
   public static final int ALICE=1, BOB=2, CECILY=3, DAVID=4;

   // Saves our class-specific data when we persist
   static class Registrations implements Serializable {
       Vector alice;
       Vector bob;
       Vector cecily;
       Vector david;
       long nextEventType;
       long lastEventSeqNo;

       Registrations(Vector alice, Vector bob, Vector cecily,
                     Vector david, long nextEventType, long lastEventSeqNo)
{
           this.alice=alice;
           this.bob=bob;
           this.cecily=cecily;
           this.david=david;
           this.nextEventType = nextEventType;
           this.lastEventSeqNo = lastEventSeqNo;
       }
   }

   // We send particular subclasses of RemoteEvent
   public static class PeopleEvent extends RemoteEvent
       implements Serializable {
       int sender;

       PeopleEvent(Object proxy, long eventType, long seq,
                    MarshalledObject data, int sender) {
           super(proxy, eventType, seq, data);
           this.sender=sender;
       }
       public int getSender() {
           return sender;
       }
   }

// Callers speak this interface to ask us to send heartbeats
   public interface PeopleRequest extends Remote {
       public EventRegistration register(int sender,
                                         MarshalledObject data,
                                         RemoteEventListener lis-
tener,
                                         long duration)
```

```
            throws RemoteException;
    }

    public PeopleGenerator(String storageLoc) throws Remote-
Exception {
        super(storageLoc);

        aliceLandlord = new SimpleEvtRegLandlord(alice, factory);
        bobLandlord = new SimpleEvtRegLandlord(bob, factory);
        cecilyLandlord = new SimpleEvtRegLandlord(cecily, factory);
        davidLandlord = new SimpleEvtRegLandlord(david, factory);
    }

    // Initialize the superclass and start the
    // leasing thread.
    protected void initialize() throws IOException, ClassNot-
FoundException {
    super.initialize();
    new Thread(this).start();
    }


    // Save the specific registration data
    protected void save() throws IOException {
        save(new Registrations(alice, bob, cecily, david,
                    nextEventType, lastEventSeqNo));
    }

    protected Object getProxy() {
    return this;
    }

    protected Entry[] getAttributes(){
    return new Entry[0];
    }


    // Restore class-specific data.
    public void restored(Object subclassData) {
    Registrations regs = (Registrations) subclassData;
        alice = regs.alice;
        bob = regs.bob;
        cecily = regs.cecily;
        david = regs.david;
        nextEventType = regs.nextEventType;
        lastEventSeqNo = regs.lastEventSeqNo;
```

```
   }

   public synchronized EventRegistration register(int sender,
                                     MarshalledObject data,
                                     RemoteEventListener lis-
tener,
                                     long duration) {
       // Build a cookie based on the table and token
   long eventType = nextEventType++;
       PeopleCookie cookie = new PeopleCookie(sender, eventType);

       // The landlord we'll use for the lease.
       SimpleEvtRegLandlord landlord;

       // The registration vector we'll add it to.
       Vector regs;

       switch (sender) {
       case ALICE:
           landlord = aliceLandlord;
           regs = alice;
           break;
       case BOB:
           landlord = bobLandlord;
           regs = bob;
           break;
       case CECILY:
           landlord = cecilyLandlord;
           regs = cecily;
           break;
       case DAVID:
           landlord = davidLandlord;
           regs = david;
           break;
       default:
           throw new IllegalArgumentException("PeopleGenerator: Bad
sender");
       }

       // Create the lease and registration, and add the
       // registration to the appropriate list.
       long expiration = landlord.getExpiration(duration);
       Registration reg = new Registration(cookie, listener,
                                           data, expiration);
       Lease lease = factory.newLease(cookie, landlord, expiration);
       regs.addElement(reg);
```

```
        // Return an event registration to the client.
        EventRegistration evtreg =
            new EventRegistration(eventType, proxy, lease, 0);

        try {
            save();
        } catch(IOException ex) {
            System.err.println("PeopleGenerator: Error check-
pointing: " +
                               ex.getMessage());
        }

        return evtreg;
    }

    protected void sendPeople() {
        tickCount++;
    if (tickCount % 7 == 0){
        sendPeoples(alice, ALICE);
        }
        if (tickCount % 4 == 0) {
            sendPeoples(bob, BOB);
        }
        if (tickCount % 5 == 0) {
            sendPeoples(cecily, CECILY);
        }
        if (tickCount % 6 == 0) {
            sendPeoples(david, DAVID);
        }
    }

    protected void sendPeoples(Vector regs, int sender) {
        // First, scavenge the list for dead registrations, in
        // reverse order (to make us immune from compaction)
        long now = System.currentTimeMillis();
        for (int i=regs.size()-1 ; i >= 0 ; i--) {
            Registration reg = (Registration) regs.elementAt(i);
            if (reg.expiration < now) {
                regs.removeElementAt(i);
            }
        }
        // Now, message the remaining listeners
        for (int i=0, size=regs.size() ; i<size ; i++) {
            Registration reg = (Registration) regs.elementAt(i);
        PeopleCookie cookie =
```

```
        (PeopleCookie) reg.getCookie();
        long eventType = cookie.eventType;

            try {
                PeopleEvent ev = new PeopleEvent(proxy,
                                        eventType,
                                                        lastEventSe-
qNo++,
                                                        reg.data, sender);
                reg.listener.notify(ev);
            } catch (RemoteException ex) {
                System.err.println("PeopleGenerator:Error no-
tifying remote
listener: " +
                                        ex.getMessage());
            } catch (UnknownEventException ex) {
                System.err.println("PeopleGenerator:Unknown event, dropping:
" +
                                        ex.getMessage());
        reg.expiration = 0;
            }
        }
    System.out.println("PeopleGenerator: Event sent, sender was "+sender);
    }

    public void run() {
        long timeToSleep =3 * 1000;
        while (true) {
            long nextWakeup = System.currentTimeMillis() + timeToSleep;
            try {
                Thread.sleep(timeToSleep);
            } catch (InterruptedException ex) {
            }
            long currentTime = System.currentTimeMillis();
            // see if we're at the next wakeup time
            if (currentTime >= nextWakeup) {
                nextWakeup = currentTime + (1000);
                // notify
                sendPeople();
            }
            timeToSleep = nextWakeup - System.currentTimeMillis();
        }
    }

    public static void main(String[] args) {
        try {
```

```
            if (args.length != 1) {
                System.err.println("Usage: PeopleGenerator " +
                                   "<storageloc>");
                System.exit(1);
            }

            PeopleGenerator gen = new PeopleGenerator(args[0]);
            gen.initialize();
            System.out.println("People generator: Started.");
        } catch (Exception ex) {
            System.err.println("PeopleGenerator: Error start-
ing people
generator: " +
                               ex.getMessage());
            System.exit(1);
        }
    }
}
```

A.3 Class SimpleEvtRegLandlord: A landlord to manage listener registrations

```
// ==============================
// *** Class SimpleEvtRegLandlord ***
// ==============================
// Usage:
// This class defines a simple Landlord implementation.
// It will manage the leases of registered listeners on be-
half of
// the various services of the jiniproject package
// ==============================
// Comments:
// This class extends UnicastRemoteObject and shall there-
fore be
// "rmic"-ed before being used.
// ==============================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ==============================

package jiniproject;


// The following packages and classes or interfaces will be used:
```

```
import net.jini.core.lease.*;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.Vector;
import java.util.Map;
import java.util.HashMap;


// As SimpleEvtRegLandlord is a subclass of UnicastRemoteOb-
ject, it
// will be necessary to "rmic"  SimpleEvtRegLandlord.
public class SimpleEvtRegLandlord extends UnicastRemoteObject
       implements Landlord {

   // The longest lease the landlord can grant (15 min by default)
   protected int maxLease = 1000*60*2;
   // Vector or cookies corresponding to registered listeners
   protected Vector regs;
   // A factory for making landlord leases.
   protected LandlordLease.Factory fac;


   public SimpleEvtRegLandlord(Vector regs, LandlordLease.Factory fac)
       throws RemoteException {
       this.regs = regs;
       this.fac = fac;
   }

   // Cancel the lease represented by cookie
   public void cancel(Object cookie) throws UnknownLeaseEx-
ception {
       for (int i=0, size=regs.size() ; i<size ; i++) {
           Registration reg = (Registration) regs.elementAt(i);
           if (reg.cookie.equals(cookie)) {
               regs.removeElementAt(i);
               return;
           }
       }
       throw new UnknownLeaseException(cookie.toString());
   }

   // Renew the lease specified by 'cookie'
   public long renew(Object cookie, long extension) throws
UnknownLeaseException {
```

```
        for (int i=0, size=regs.size() ; i<size ; i++) {
            Registration reg = (Registration) regs.elementAt(i);
            if (reg.getCookie().equals(cookie)) {
                long expiration = getExpiration(extension);
                reg.setExpiration(expiration);
                return expiration - System.currentTimeMillis();
            }
        }
        throw new UnknownLeaseException(cookie.toString());
    }

    // Cancel a set of leases (note the declaration of this method has
    // changed since jini 1.0, and this implementation will only work
    // on jini 1.1. Should return null if everything is fine.
    public Map cancelAll(Object[] cookies) throws RemoteException{
        Map exceptionMap = null;
        for (int i=0 ; i<cookies.length ; i++) {
            try {
                cancel(cookies[i]);
            }
        catch (UnknownLeaseException ex) {
                if (exceptionMap == null) {
                    exceptionMap = new HashMap();
                }
                Lease lease = fac.newLease(cookies[i], this, 0);
                exceptionMap.put(lease, ex);
            }
        }
    return(exceptionMap);
    }

    // Renew a set of leases. If everything is fine, denied should be null.
    public Landlord.RenewResults renewAll(Object[] cookies, long[]
extensions) {
        long[] granted = new long[cookies.length];
        Exception[] denied = null;
        for (int i=0 ; i<cookies.length ; i++) {
            try {
                granted[i] = renew(cookies[i], extensions[i]);
            }
        catch (Exception ex) {
                if (denied == null) {
                    denied = new Exception[cookies.length+1];
                }
                denied[i+1] = ex;
            }
```

```
        }
        Landlord.RenewResults results =
new Landlord.RenewResults(granted, denied);
        return results;
    }

    // A trivial policy to determine the next expiration time when
    // a listener requests a new lease:
    // If the requested time lease is longer than maxLease, a new
    // lease of a duration maxLease wil be granted. Else, the requested
    // lease is granted.
    public long getExpiration(long request) {
        if (request > maxLease || request == Lease.ANY)
            return System.currentTimeMillis() + maxLease;
        else
            return System.currentTimeMillis() + request;
    }

    // Allows services to set a new value for the longest lease.
    public void setMaxLease(int maxLease) {
        this.maxLease = maxLease;
    }
}
```

A.4 Class Registration: An object to store data related to one listener

```
// ==============================
// Class Registration
// ==============================
// Usage:
// This class defines a registration, that is actually the object
// a SimpleEvtLandlord uses to keep data about listeners.
// ==============================
// Requirements:
// Jini 1.0 or higher, Java 1.2 or higher
// ==============================
package jiniproject;

import java.io.Serializable;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEventListener;

class Registration implements Serializable {
```

```
    protected Object cookie;
    protected RemoteEventListener listener;
    protected MarshalledObject data;
    protected long expiration;

    // To maintain a registration we need to remember
    // the cookie for the registration, who the listener
    // is, the client-provided data, and its expiration
    // time.
    Registration(Object cookie, RemoteEventListener listener,
                 MarshalledObject data, long expiration) {
        this.cookie = cookie;
        this.listener = listener;
        this.data = data;
        this.expiration = expiration;
    }

    Object getCookie() {
        return cookie;
    }
    RemoteEventListener getListener() {
        return listener;
    }
    MarshalledObject getData() {
        return data;
    }
    long getExpiration() {
        return expiration;
    }
    void setExpiration(long expiration) {
        this.expiration = expiration;
    }
}
```

A.5 Class PeopleClient: A listener for non-ordered events

```
// ==============================
// *** Class PeopleClient ***
// ==============================
// Usage:
// A simple service that receives the events directly from
// PeopleGenerator (i.e. before they are ordered by EventOrderer)
// ==============================
// Nested classes:
// -Class DiscListener
```

```
// -Class EvtListener
// ==============================
// Comments:
// The nested class EvtListener needs to be "rmic"-ed as it extends
// UnicastRemoteObject
// ==============================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ==============================

package jiniproject;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;
import net.jini.lease.LeaseRenewalManager;
import com.sun.jini.lease.LeaseRenewalEvent;
import com.sun.jini.lease.LeaseListener;
import net.jini.core.lease.Lease;
import java.io.IOException;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class PeopleClient implements Runnable {
   ServiceTemplate template;
   PeopleGenerator.PeopleRequest
       gen = null;
   LeaseRenewalManager leaseManager =
       new LeaseRenewalManager();
   EvtListener listener;

   // An inner class for discovery listening
   public class DiscListener implements DiscoveryListener {
       public void discovered(DiscoveryEvent ev) {
           ServiceRegistrar[] regs = ev.getRegistrars();
           for (int i=0 ; i<regs.length ; i++) {
               doit(regs[i]);
           }
       }
       public void discarded(DiscoveryEvent ev) {
```

```
    }
}

// an inner class for event listening
class EvtListener extends UnicastRemoteObject
    implements RemoteEventListener {
    EvtListener() throws RemoteException {
    }
    public void notify(RemoteEvent e) {
        if (!(e instanceof PeopleGenerator.PeopleEvent)) {
            return;
        }
        PeopleGenerator.PeopleEvent ev =
            (PeopleGenerator.PeopleEvent) e;

        switch (ev.getSender()) {
        case PeopleGenerator.ALICE:
            System.out.println("PeopleClient: Came from ALICE");
            break;
        case PeopleGenerator.BOB:
            System.out.println("PeopleClient: Came from BOB");
            break;
        case PeopleGenerator.CECILY:
            System.out.println("PeopleClient: Came from CECILY");
            break;
        case PeopleGenerator.DAVID:
            System.out.println("PeopleClient: Came from DAVID");
            break;
        default:
            System.out.println("PeopleClient: Unknown!");
            break;
        }
    }
}

public PeopleClient() throws IOException, RemoteException {
    Class[] types = { PeopleGenerator.PeopleRequest.class };
    template = new ServiceTemplate(null, types, null);
    listener = new EvtListener();
    LookupDiscovery disco =
        new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    disco.addDiscoveryListener(new DiscListener());
}

void doit(ServiceRegistrar reg) {
    if (gen != null)
```

```
            return;
        try {
            gen = (PeopleGenerator.PeopleRequest)
                reg.lookup(template);
        }
    catch (Exception ex) {
            System.err.println("PeopleClient: Doing lookup: " +
ex.getMessage());
        }
        if (gen == null)
            return;
        EventRegistration evt;
        try {
            evt = gen.register(PeopleGenerator.ALICE, null, lis-
tener,
Lease.ANY);
            leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
            evt = gen.register(PeopleGenerator.BOB, null, lis-
tener,
Lease.ANY);
            leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
            evt = gen.register(PeopleGenerator.CECILY, null, lis-
tener,
Lease.ANY);
            leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
            evt = gen.register(PeopleGenerator.DAVID, null, lis-
tener,
Lease.ANY);
            leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
        }
    catch (RemoteException ex) {
            System.err.println("PeopleClient: Bogus: " + ex.getMessage());
        }
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(Integer.MAX_VALUE);
            }
        catch (InterruptedException ex) {
            }
        }
    }

    public static void main(String[] args) {
```

```
        try {
        System.out.println("PeopleClient: Building...");
            System.setSecurityManager(new RMISecurityManager());
            PeopleClient client = new PeopleClient();
        System.out.println("PeopleClient: Started");
            new Thread(client).start();
        }
    catch (Exception ex) {
            System.err.println("PeopleClient: Bogus: " + ex.getMessage());
            System.exit(1);
        }
    }
}
```

A.6 Class EventOrderer: The actual event ordering service

```
// ==============================
// *** Class EventOrderer ***
// ==============================
// Usage:
// This class defines an orderer that will forward and tag the
// events it receives by nesting them in a specific type of events
// It consists mainly in a client part that will register with the
// event generator(s), and in a generator part, that will create
// and send the new events to the registered listeners.
// Ordering events is achiveved by increasing sequence num-
ber of
// the new events.
// ==============================
// Nested classes:
// -Class OrderedCookie
// -Class OrderedEvent
// -Interface EventOrdererRequest
// ==============================
// Comments:
// This class extends BasicService and shall therefore be
// "rmic"-ed before being used.
// ==============================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ==============================

package jiniproject;
```

```
// The needed packages and classes:
import net.jini.core.lease.*;
import net.jini.core.discovery.*;
import net.jini.core.lookup.*;
import net.jini.core.event.*;
import net.jini.lease.*;
import net.jini.discovery.*;
import net.jini.lookup.*;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import net.jini.core.entry.Entry;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.io.IOException;
import java.io.Serializable;
import java.util.Vector;
import java.util.Hashtable;


// Nested class defining a new type of Cookies that should be associated
// with the registrations
class OrderedCookie implements Serializable {
   protected long eventType;
   OrderedCookie(long eventType) {
       this.eventType = eventType;
   }
   public boolean equals(Object o) {
       if (!(o instanceof OrderedCookie))
           return false;
       OrderedCookie cookie = (OrderedCookie) o;
       return cookie.eventType == eventType;
   }
}

// EventOrderer is a BasicService
public class EventOrderer extends BasicService
   implements Runnable, EventOrderer.EventOrdererRequest,
          RemoteEventListener, DiscoveryListener {

   // Generator-side variables
   protected static final int MAX_LEASE = 1000 * 60 * 10;
   protected LandlordLease.Factory fac = new LandlordLease.Factory();
   protected long lastEventSeqNo = 0;
   protected Vector regs = new Vector();
   protected long nextEventType = 2179;
```

```
   protected SimpleEvtRegLandlord landlord;

   // Client-side variables
   protected ServiceTemplate template;
   protected PeopleGenerator.PeopleRequest gen = null;
   protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

   // OrderedEvent is a subclass of RemoteEvent that can nest
   // another RemoteEvent
   public static class OrderedEvent extends RemoteEvent
       implements Serializable {

       RemoteEvent rev;
   // We use the source and data object that are in the re-
ceived event
   // in the new event.
   OrderedEvent(long eventType, long seq, RemoteEvent rev) {
            super(rev.getSource(), eventType, seq,
rev.getRegistrationObject());
            this.rev=rev;
       }
   // This method will allow clients to access the nested event
       public RemoteEvent getOriginalEvent() {
           return rev;
       }
   }

   // Defines the way clients have to register with this service
   public interface EventOrdererRequest extends Remote {
       public EventRegistration register(MarshalledObject data,
                     RemoteEventListener listener,long duration)
           throws RemoteException;
   }

   protected EventOrderer() throws RemoteException {
       super("/tmp/eolog");

   //Generator-side
   landlord=new SimpleEvtRegLandlord(regs, fac);

   //Client-side
   Class[] types = { PeopleGenerator.PeopleRequest.class };
       template = new ServiceTemplate(null, types, null);
   try{
       LookupDiscovery disco = new
LookupDiscovery(LookupDiscovery.ALL_GROUPS);
```

```
        disco.addDiscoveryListener(this);
    }
    catch(IOException ex){
        System.err.println("EventOrderer: Error during LookupDiscovery");
    }
    }

    // DiscoveryListener methods:
    public void discovered(DiscoveryEvent ev) {
            ServiceRegistrar[] regs = ev.getRegistrars();
            for (int i=0 ; i<regs.length ; i++) {
                doit(regs[i]);
            }
        }
    public void discarded(DiscoveryEvent ev) {
        }

    // Will call the forward(Event) method when an interest-
ing event arrives
    public void notify(RemoteEvent e) {
    if (!(e instanceof PeopleGenerator.PeopleEvent)) {
        return;
    }
    PeopleGenerator.PeopleEvent ev =
        (PeopleGenerator.PeopleEvent) e;
    switch (ev.getSender()) {
    case PeopleGenerator.ALICE:
        System.out.println("EventOrderer: Came from Alice");
        break;
    case PeopleGenerator.BOB:
        System.out.println("EventOrderer: Came from Bob");
        break;
    case PeopleGenerator.CECILY:
        System.out.println("EventOrderer: Came from Cecily");
        break;
    case PeopleGenerator.DAVID:
        System.out.println("EventOrderer: Came from David");
        break;
    default:
        System.out.println("EventOrderer: Unknown!");
        break;
    }
    System.out.println("EventOrderer: Will forward...");
    forward(ev);
    }
```

```
   // Register with the generators
   protected void doit(ServiceRegistrar reg) {
       if (gen != null)
           return;
       try {
           gen = (PeopleGenerator.PeopleRequest)reg.lookup(template);
       } catch (Exception ex) {
           System.err.println("EventOrderer: Doing lookup: " +
ex.getMessage());
       }
       if (gen == null)
           return;
       EventRegistration evt;
       try {
           evt = gen.register(PeopleGenerator.ALICE, null, this, Lease.ANY);
           leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
           evt = gen.register(PeopleGenerator.BOB, null, this, Lease.ANY);
           leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
           evt = gen.register(PeopleGenerator.CECILY, null, this,
Lease.ANY);
           leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
           evt = gen.register(PeopleGenerator.DAVID, null, this, Lease.ANY);
           leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
       }
   catch (Exception ex) {
       if (ex instanceof RemoteException)
       System.err.println("EventOrderer: Bogus: " + ex.getMessage());
       }
   }

   // Creates a new event that nests the received event
   protected void forward(RemoteEvent ev){
       // First, scavenge the list for dead registrations, in
       // reverse order (to make us immune from compaction)
       long now = System.currentTimeMillis();
       for (int i=regs.size()-1 ; i >= 0 ; i--) {
           Registration reg = (Registration) regs.elementAt(i);
           if (reg.expiration < now) {
               regs.removeElementAt(i);
           }
       }
       // Now, forward event to the remaining listeners
       for (int i=0, size=regs.size() ; i<size ; i++) {
           Registration reg = (Registration) regs.elementAt(i);
       OrderedCookie cookie = (OrderedCookie) reg.getCookie();
       long eventType = cookie.eventType;
```

```
           try {
               OrderedEvent oev = new OrderedEvent(eventType,
lastEventSeqNo++, ev);
               reg.listener.notify(oev);
               }
       catch (RemoteException ex) {
               System.err.println("EventOrderer: Error no-
tifying remote
listener: " +
                                   ex.getMessage());
               }
       catch (UnknownEventException ex) {
               System.err.println("EventOrderer: Unknown event, drop-
ping: "
+
                                   ex.getMessage());
       reg.expiration = 0;
               }
       }
   System.out.println("EventOrderer: Event forwarded to "+regs.size()+"
clients");
   }

   // This method is called by clients that want to listen to the ordered
events
   public synchronized EventRegistration register(MarshalledObject data,
           RemoteEventListener listener,long duration){
   long eventType = nextEventType++;
       OrderedCookie cookie = new OrderedCookie(eventType);
       SimpleEvtRegLandlord landlord = this.landlord;
       Vector regs = this.regs;
   long expiration = landlord.getExpiration(duration);
       Registration reg = new Registration(cookie, listener, data,
expiration);
       Lease lease = fac.newLease(cookie, landlord, expiration);
       regs.addElement(reg);
       EventRegistration evtreg = new EventRegistration(eventType, proxy,
lease, 0);
   System.out.println("EventOrderer: New registration completed");
       return evtreg;
   }

   public void initialize() throws IOException, ClassNotFoundEx-
ception {
       super.initialize();
       new Thread(this).start();
```

```
   }

   // This object is actually its own proxy and can be used by clients
through the
   // nested EventOrdererRequest interface
   protected Object getProxy() {
       return this;
   }
   protected Entry[] getAttributes(){
   return new Entry[0];
   }

   public void run() {
       while (true) {
           try {
               Thread.sleep(Integer.MAX_VALUE);
           } catch (InterruptedException ex) {
           }
       }
   }

   //Remember to always set a security manager !!!
   public static void main(String[] args) {
       try {
       System.setSecurityManager(new RMISecurityManager());
       System.out.println("EventOrderer: Building...");
           EventOrderer eo = new EventOrderer();
       System.out.print("EventOrderer: Intializing...");
           eo.initialize();
           System.out.println("EventOrderer: Running!");
       } catch (Exception ex) {
           System.err.println("EventOrderer: Error starting event or-
derer: "
+
                              ex.getMessage());
           System.exit(1);
       }
   }
}
```

A.7 Class OrderedPeopleClient: A listener for ordered events

```
// ==============================
```

```
// *** Class OrderedPeopleClient ***
// ================================
// Usage:
// A simple service that receives the events after they have been
// ordered by EventOrderer
// ================================
// Nested classes:
// -Class DiscListener
// -Class EvtListener
// ================================
// Comments:
// The nested class EvtListener needs to be "rmic"-ed as it extends
// UnicastRemoteObject
// ================================
// Requirements:
// Jini 1.1, Java 1.2 or higher
// ================================


package jiniproject;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEventListener;
import net.jini.lease.LeaseRenewalManager;
import com.sun.jini.lease.LeaseRenewalEvent;
import com.sun.jini.lease.LeaseListener;
import net.jini.core.lease.Lease;
import java.io.IOException;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;


public class OrderedPeopleClient implements Runnable {
   ServiceTemplate template;
   EventOrderer.EventOrdererRequest gen = null;
   LeaseRenewalManager leaseManager = new LeaseRenewalManager();
   EvtListener listener;

   public class DiscListener implements DiscoveryListener {
```

```
        public void discovered(DiscoveryEvent ev) {
            ServiceRegistrar[] regs = ev.getRegistrars();
            for (int i=0 ; i<regs.length ; i++) {
                doit(regs[i]);
            }
        }
        public void discarded(DiscoveryEvent ev) {
        }
    }

    class EvtListener extends UnicastRemoteObject implements
RemoteEventListener {

    EvtListener() throws RemoteException {
        }

    public void notify(RemoteEvent e) {
            if (!(e instanceof EventOrderer.OrderedEvent)) {
        if (!(e instanceof PeopleGenerator.PeopleEvent))
            System.out.println("OrderedPeopleClient: Event wasn't
recognized");
                return;
            }
        EventOrderer.OrderedEvent oev = (EventOrderer.OrderedEvent) e;
        RemoteEvent rev = oev.getOriginalEvent();
        if (!(rev instanceof PeopleGenerator.PeopleEvent)) {
        System.out.println("OrderedPeopleClient: Nested event wasn't
recognized");
                return;
            }
            PeopleGenerator.PeopleEvent ev = (PeopleGenerator.PeopleEvent)
rev;
            switch (ev.getSender()) {
            case PeopleGenerator.ALICE:
                System.out.println("OrderedPeopleClient: Came from AL-
ICE,
position="+e.getSequenceNumber());
                break;
            case PeopleGenerator.BOB:
                System.out.println("OrderedPeopleClient: Came from BOB,
position="+e.getSequenceNumber());
                break;
            case PeopleGenerator.CECILY:
                System.out.println("OrderedPeopleClient: Came from CE-
CILY,
position="+e.getSequenceNumber());
```

```
                break;
            case PeopleGenerator.DAVID:
                System.out.println("OrderedPeopleClient: Came from DAVID,
position="+e.getSequenceNumber());
                break;
            default:
                System.out.println("OrderedPeopleClient: Unknown!");
                break;
            }
        }
    }

    public OrderedPeopleClient() throws IOException, Remote-
Exception {
        Class[] types = { EventOrderer.EventOrdererRequest.class };
        template = new ServiceTemplate(null, types, null);
        listener = new EvtListener();
        LookupDiscovery disco = new
LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        disco.addDiscoveryListener(new DiscListener());
    }

    void doit(ServiceRegistrar reg) {
        if (gen != null)
            return;
        try {
            gen = (EventOrderer.EventOrdererRequest)reg.lookup(template);
        }
    catch (Exception ex) {
            System.err.println("OrderedPeopleClient: Doing lookup: " +
ex.getMessage());
        }
        if (gen == null)
            return;
        EventRegistration evt;
        try {
            evt = gen.register(null, listener, Lease.ANY);
            leaseManager.renewUntil(evt.getLease(), Lease.ANY, null);
    }
    catch (RemoteException ex) {
            System.err.println("OrderedPeopleClient: Bogus: " +
ex.getMessage());
        }
    }

    public void run() {
```

```
        while (true) {
            try {
                Thread.sleep(Integer.MAX_VALUE);
            }
        catch (InterruptedException ex) {
            }
        }
    }

    public static void main(String[] args) {
        try {
        System.out.println("OrderedPeopleClient: Building...");
            System.setSecurityManager(new RMISecurityManager());
            OrderedPeopleClient client = new OrderedPeopleClient();
        System.out.println("OrderedPeopleClient: Started");
            new Thread(client).start();
        } catch (Exception ex) {
            System.err.println("OrderedPeopleClient: Bogus: " +
ex.getMessage());
            System.exit(1);
        }
    }
}
```

Appendix B

Some tips to get started with Jini

Here are the first steps:

1. Acquire some basic knowledge on Java and RMI.
2. Download Sun's implementation of Jini (http://java.sun.com).
3. Try to start the example services (often takes a long time!).
4. Subscribe to some Jini-users newsgroups to get some help (http://www.jini.org).
5. Find some documentation about the last version of Jini.
6. Develop a simple service to understand the mechanisms.

And here are the things a Jini beginner should know:

- Jini requires Java 1.2 or higher. On some Linux releases as well as on MacOS XS public beta, JVM are bogged. On Linux, it is possible to change JVM, and on MacOS, Apple say they will fix the bugs (one day...).
- Java 1.2 UnicastRemoteObject 's always need to be "rmic"-ed.
- Services that were developed to work with Jini 1.0 might not work with 1.1.

And eventually here are the things that a Jini programmer should never forget:

- Always set a security manager when running rmi objects (unless it will not work).
- Never forget to start rmid and rmiregistry when required.
- Be careful when developing on a single machine, as paths could be locally valid but wrong on the network.

Bibliography

[1] The community resource for jini technologie. This website provides useful information about Jini (http://www.jini.org).

[2] Keith Edwards. *Core JINI*. PrencticeHall, Upper Saddle River, New Jersey, USA, 1999.

[3] Sing Li. *Professional JINI*. Wrox Press, 1999.

[4] Scott Oaks and Henry Wong. *JINI in a nutshell*. O'Reilly, Morris Street, Sebastopol, California, USA, 2000.

[5] Sun microsystems. *JINI specifications v 1.1*, 2000. (Downloadable at http://www.sun.com/jini/specs/).